such as a printer, or a device that can have both input and output operations, such as a disk drive. This is illustrated in Figure 10-1.

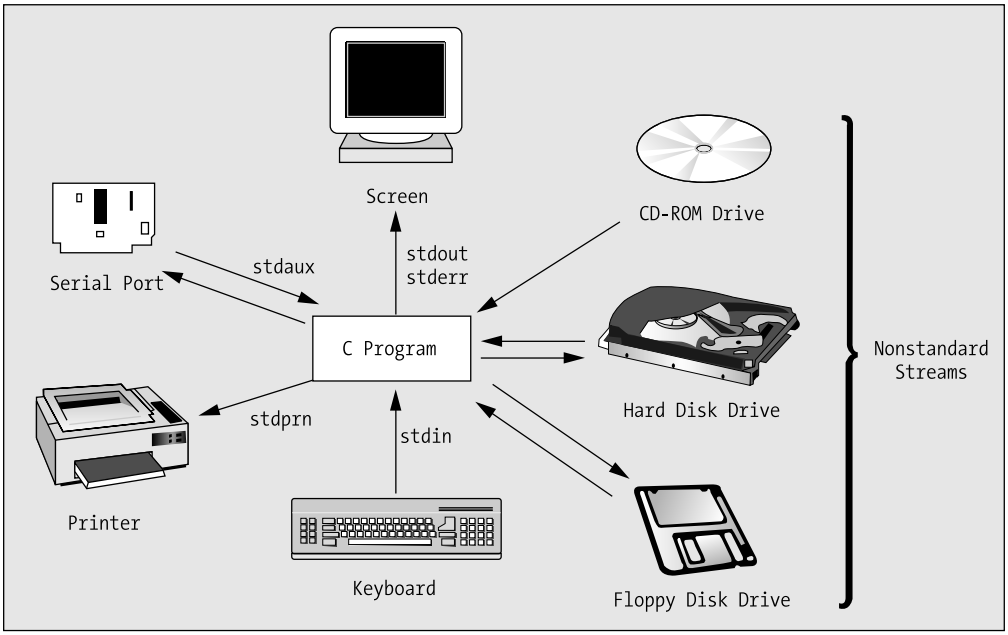


Figure 10-1. *Standard and nonstandard streams*

A disk drive can have multiple input and output streams because it can contain multiple files. The correspondence is between a stream and a file, not between a stream and the device. A stream can be associated with a specific file on the disk. The stream that you associate with a particular file could be an input stream, so you could only read from the file; it could be an output stream, in which case you could only write to the file; or the stream might allow input and output so reading and writing the file would both be possible. Obviously, if the stream that is associated with a file is an input stream, the file must have been written at some time so it contained some data. You could also associate a stream with a file on a CD-ROM drive. Because this device is typically read-only, the stream would, of necessity, be an input stream.

There are two further kinds of streams: **character streams**, which are also referred to as **text streams**, and **binary streams**. The main difference between these is that data transferred to or from character streams is treated as a sequence of characters and may be modified by the library routine concerned, according to a format specification. Data that's transferred to or from binary streams is just a sequence of bytes that isn't modified in any way. I discuss binary streams in Chapter 12 when I cover reading and writing disk files.

Standard Streams

C has three predefined **standard streams** that are automatically available in any program, provided, of course, that you've included the `<stdio.h>` header file, which contains their definitions, into your program. These standard streams are `stdin`, `stdout`, and `stderr`. Two other streams that are available

with some systems are identified by the names `stdprn` and `stdaux`, but these are not part of the C language standard so your compiler may not support them.

No initialization or preparation is necessary to use these streams. You just have to apply the appropriate library function that sends data to them. They are each preassigned to a specific physical device, as shown in Table 10-1.

Table 10-1. *Standard Streams*

Stream	Device
`stdin`	Keyboard
`stdout`	Display screen
`stderr`	Display screen
`stdprn`	Printer
`stdaux`	Serial port

In this chapter I concentrate on how you can use the standard input stream, `stdin`, the standard output stream, `stdout`, and the printer stream, `stdprn`.

The `stderr` stream is simply the stream to which error messages from the C library are sent, and you can direct your own error messages to `stderr` if you wish. The main difference between `stdout` and `stderr` is that output to `stdout` is buffered in memory so the data that you write to it won't necessarily be transferred immediately to the device, whereas `stderr` is unbuffered so any data you write to it is always transferred immediately to the device. With a buffered stream your program transfers data to or from a buffer area in memory, and the actual data transfer to or from the physical device can occur asynchronously. This makes the input and output operations much more efficient. The advantage of using an unbuffered stream for error messages is that you can be sure that they will actually be displayed but the output operations will be inefficient; a buffered stream is efficient but may not get flushed when a program fails for some reason, so the output may never be seen. I won't discuss this further, other than to say `stderr` points to the display screen and can't be redirected to another device. Output to the stream `stdaux` is directed to the serial port and is outside the scope of this book for reasons of space rather than complexity.

Both `stdin` and `stdout` can be reassigned to files, instead of the default of keyboard and screen, by using operating system commands. This offers you a lot of flexibility. If you want to run your program several times with the same data, during testing for example, you could prepare the data as a text file and redirect `stdin` to the file. This enables you to rerun the program with the same data without having to re-enter it each time. By redirecting the output from your program to a file, you can easily retain it for future reference, and you could use a text editor to access it or search it.

Input from the Keyboard

There are two forms of input from the keyboard on `stdin` that you've already seen in previous chapters: **formatted input**, which is provided primarily by the `scanf()` function and **unformatted input**, in which you receive the raw character data from a function such as `getchar()`. There's rather more to both of these possibilities, so let's look at them in detail.

Formatted Keyboard Input

As you know, the function scanf() reads characters from the stream stdin and converts them to one or more values according to the format specifiers in a format control string. The prototype of the scanf() function is as follows:

```
int scanf(char *format, ... );
```

The format control string parameter is of type char *, a pointer to a character string as shown here. However, this usually appears as an explicit argument in the function call, such as

```
scanf("%lf", &variable);
```

But there's nothing to prevent you from writing this:

```
char str[] = "%lf";
scanf(str, &variable);
```

The scanf() function makes use of the facility of handling a variable number of arguments that you learned about in Chapter 9. The format control string is basically a coded description of how scanf() should convert the incoming character stream to the values required. Following the format control string, you can have one or more optional arguments, each of which is an address in which a corresponding converted input value is to be stored. As you've seen, this implies that each of these arguments must be a pointer or a variable name prefixed by & to define the address of the variable rather than its value.

The scanf() function reads from stdin until it comes to the end of the format control string, or until an error condition stops the input process. This sort of error is the result of input that doesn't correspond to what is expected with the current format specifier, as you'll see. Something that I haven't previously noted is that scanf() returns a value that is the count of the number of input values read. This provides a way for you to detect when an error occurs by comparing the value returned by scanf() with the number of input values you are expecting.

The wscanf() function provides exactly the same capabilities as scanf() except that the first argument to the function, which is the format control string, must be a wide character string of type wchar_t *.

Thus you could use wscanf() to read a floating-point value from the keyboard like this:

```
wscanf(L"%lf", &variable);
```

The first argument is a wide character string constant and in all other respects the function works like scanf(). If you omit the L for the wide character string literal, you will get an error message from the compiler because your argument does not match the type of the first parameter.

Of course, you could also write this:

```
wchar_t wstr[] = L"%lf";
wscanf(wstr, &variable);
```

Input Format Control Strings

The format control string that you use with scanf() or wscanf() isn't precisely the same as that used with printf(). For one thing, putting one or more whitespace characters—blank ' ', tab '\t', or newline '\n'—in the format control string causes scanf() to read and ignore whitespace characters up to the next nonwhitespace character in the input. A single whitespace character in the format control string causes any number of consecutive whitespace characters to be ignored. You can therefore include as many whitespace characters as you wish in the format string to make it more readable. Note that whitespace characters are ignored by scanf() by default except when you are reading data using %c, %[], or %n specifications (see Table 10-2).

Any nonwhitespace character other than % will cause scanf() to read but not store successive occurrences of the character. If you want scanf() to ignore commas separating values in the input for instance, just precede each format specifier by a comma. There are other differences too, as you'll see when I discuss formatted output in the section "Output to the Screen" a bit later in this chapter.

The most general form of a format specifier is shown in Figure 10-2.

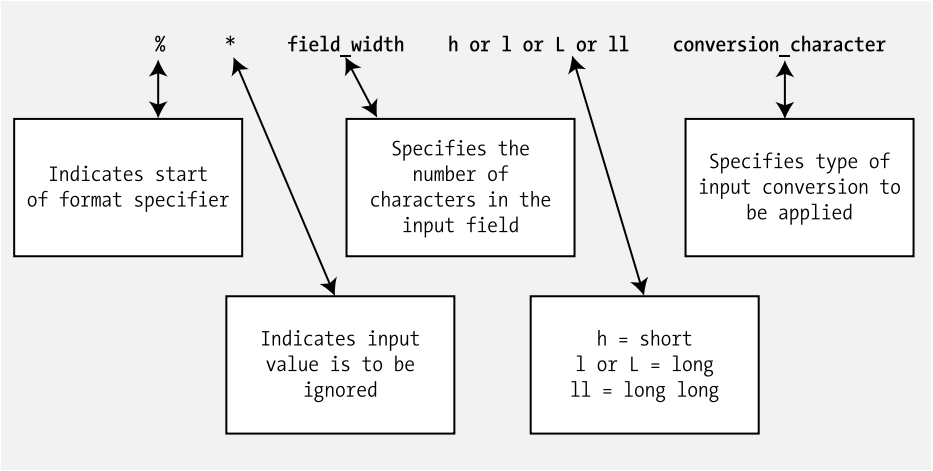


Figure 10-2. *The general form of an output specifier*

Let's take a look at what the various parts of this general form mean:

- The % simply indicates the start of the format specifier. It must always be present.
- The next * is optional. If you include it, it indicates that the next input value is to be ignored. This isn't normally used with input from the keyboard. It does become useful, however, when stdin has been reassigned to a file and you don't want to process all the values that appear within the file in your program.
- The field width is optional. It's an integer specifying the number of characters that scanf() should assume makes up the current value being input. This allows you to input a sequence of values without spaces between them. This is also often quite useful when reading files.
- The next character is also optional, and it can be h, L, l (the lowercase letter L), or ll (two lowercase Ls). If it's h, it can only be included with an integer conversion specifier (d, I, o, u, or x) and indicates that the input is to be converted as short. If it's l, it indicates long when preceding an int conversion specifier, and double when preceding a float conversion specifier. Prefixing the c specification with l specifies a wide character conversion so the input is read as wchar_t. The prefix L applied to e, E, f, g, or G specifies the value is of type long double. The ll prefix applies to integer conversions and specifies that the input is to be stored as type long long.
- The conversion character specifies the type of conversion to be carried out on the input stream and therefore must be included. The possible characters and their meanings are shown in Table 10-2.

Table 10-2. *Conversion Characters and Their Meanings*

Conversion Character	Meaning
`d`	Convert input to `int`.
`i`	Convert input to `int`. If preceded by `0`, then assume octal digits input. If preceded by `0x` or `0X`, then assume hexadecimal digits input.
`o`	Convert input to `int` and assume all digits are octal.
`u`	Convert input to `unsigned int`.
`x`	Convert to `int` and assume all digits are hexadecimal.
`c`	Read the next character as `char` (including whitespace). If you want to ignore whitespace when reading a single character, just precede the format specification by a whitespace character.
`s`	Input a string of successive nonwhitespace characters, starting with the next nonwhitespace character.
`e`, `f`, or `g`	Convert input to type `float`. A decimal point and an exponent in the input are optional.
`n`	No input is read but the number characters that have been read from the input source up to this point are stored in the corresponding argument, which should be of type `int*`.

You can also read a string that consists of specific characters by placing all the possible characters between square brackets in the specification. For example, the specification `%[0123456789.-]` will read a numerical value as a string, so if the input is –1.25 it will be read as `"-1.25"`. To read a string consisting of the lowercase letters *a* to *z*, you could use the specification `%[abcdefghijklmnopqrstuvwxyz]`. This will read any sequence of the characters that appear between the square brackets as a string, and the first character in the input that isn't in the set between the square brackets marks the end of the input. Although it isn't required by the standard, many C library implementations support the form `%[a-z]` to read a string consisting of any lowercase letters.

A specification using square brackets is also very useful for reading strings that are delimited by characters other than whitespace. In this case, you can specify the characters that are *not* in the string by using the `^` character as the first character in the set. Thus, the specification `%[^,]` will include everything in the string except a comma, so this form will enable you to read a series of strings separated by commas.

Table 10-3 shows a few examples of applying the various options.

Table 10-3. *Examples of Options in Conversion Specifications*

Specification	Description
`%lf`	Read the next value as type `double`
`%*d`	Read the next integer value but don't store it
`%lc`	Read the next character as type `wchar_t`
`%\nc`	Read the next character as type `char` ignoring whitespace characters
`%10lld`	Reads the next ten characters as an integer value of type `long long`

Table 10-3. *Examples of Options in Conversion Specifications*

Specification	Description
`%5d`	Read the next five characters as an integer
`%Lf`	Read the next value as a floating-point value of type `long double`
`%hu`	Read the next value as type `unsigned short`

Let's exercise some of these format control strings with practical examples.

TRY IT OUT: EXERCISING FORMATTED INPUT

To start, let's read a variety of data and then output the result:

```
/* Program 10.1        Exercising formatted input */
#include <stdio.h>

const size_t SIZE = 20;      /* Max characters in a word */

int main(void)
{
  int value_count = 0;       /* Count of input values read */
  float fp1 = 0.0;           /* Floating-point value read  */
  int i = 0;                 /* First integer read         */
  int j = 0;                 /* Second integer read        */
  char word1[SIZE] = " ";    /* First string read          */
  char word2[SIZE] = " ";    /* Second string read         */
  int byte_count = 0;        /* Count of input bytes read  */

  value_count = scanf("%f %d %d %[abcdefghijklmnopqrstuvwxyz] %*1d %s%n",
                              &fp1, &i , &j, word1, word2, &byte_count);
  printf("\nCount of bytes read = %d\n", byte_count);
  printf("\nCount of values read = %d\n", k);
  printf("\nfp1 = %f   i = %d   j = %d", fp1, i, j);
  printf("\nword1 = %s   word2 = %s\n", word1, word2);
  return 0;
}
```

Here's an example of the output from this program:

```
-2.35 15 25 ready2go

Count of bytes read = 20

Count of values read = 5

fp1 = -2.350000   i = 15   j = 25
word1 = ready   word2 = go
```

How It Works

The first three input values are read in a straightforward way. The fourth input value is read using the specifier "%[abcdefghijklmnopqrstuvwxyz]", which will read a sequence of lowercase letters as a string. This reads the string "ready" because the character that follows, '2', isn't in the set between the square brackets. The '2' in the input is read using the specifier "%*1d". The * in the specification causes the input to be read but not stored, and the field width is one character. The word "go" is read and stored in word2 using the "%s" specifier. The %n specifier does not extract data from the input stream; it just stores the number of bytes that the scanf() function has read from the input stream up to that point so that value is stored in byte_count.

The value of value_count holds the count of the number of values processed that is returned by the scanf() function. As you can see, the value reflects the number of values stored and doesn't include the value read by the "%*1d" specification.

It isn't essential that you enter all the data on a single line. If you key in the first two values and press Enter, the scanf() function will wait for you to enter the next value on the next line.

Now let's change the program a little bit by altering one statement. Replace the input statement with

```
value_count = scanf(
              "%4f %d %d %*d %[abcdefghijklmnopqrstuvwxyz] %*1d %[^o]%n",
              &fp1, &i , &j, word1, word2, &byte_count);
```

Now run the program with exactly the same input line as before. You should get the following output:

```
-2.35 15 25 ready2go

Count of bytes read = 19

Count of values read = 5

fp1 = -2.300000   i = 5   j = 15
word1 = ready   word2 = g
```

Because you specified a field width of 4 for the floating-point value, the first four characters are taken as defining the value of the first input variable. The following integer value to be input is read as 5, the last digit following the value read as –2.3. The integer that's stored in j is 15, and the value 25 is read and ignored by the "%*d" specification. The last string is read now as just "g" because the specification "%[^o]" accepts any character in the string except the letter 'o'. Because the letter 'o' is not read as part of the input the byte count is now 19.

Let's try another variation. Change the input statement to this:

```
value_count = scanf(
              "%4f %4d %d %*d %[abcdefghijklmnopqrstuvwxyz] %*1d %[^o]%n",
              &fp1, &i , &j, word1, word2, &byte_count);
```

With exactly the same input line as before, you should now get the following output:

```
-2.35 15 25 ready2go

Count of bytes read = 19

Count of values read = 5

fp1 = -2.300000   i = 5   j = 15
word1 = ready   word2 = g
```

So what can you conclude from this case? The first floating-point value has clearly been defined by the first four characters of input. The next two values result in the integers 5 and 15. This shows that in spite of the fact that you specified a field width of 4 for the second integer, it appears to have been overridden. This is a consequence of the blank following digit 5, which terminates the input scanning for the value being read. So whatever value you put as a field width, the scanning of the input line for a given value stops as soon as you meet the first blank. You could change the specifiers for the integer values to `%12d` and the result would still be the same for the given input.

You can demonstrate one further aspect of numerical input processing by running the last version of the previous example with a slightly different input line:

```
-2.3A 15 25 ready2go

Count of bytes read = 0

Count of values read = 1

fp1 = -2.300000   i = 0   j = 0
word1 =     word2 =
```

The count of the number of input values is 1, corresponding to a value for the variable `fp1` being read. The count of the number of bytes read is zero, which is clearly incorrect, the reason being that we never got to store a value in `byte_count`. The `A` in the input stream is invalid in numerical input, and so the whole process stops dead. No values for variables `i` and `j`, `word1` and `word2`, are processed, and no value is stored for the byte count. This demonstrates how unforgiving `scanf()` really is. A single invalid character in the input stream will stop your program in its tracks. If you want to be able to recover from the situation in which invalid input is entered, you can use the return value from `scanf()` as a measure of whether all the necessary input has been processed correctly and include some code to retrieve the situation when necessary.

The simplest approach would perhaps be to print an irritable message and then demand the whole input be repeated. But beware of errors in your code getting you into a permanent loop in this circumstance. You'll need to think through all of the possible ways that things might go wrong if you're going to produce a robust program.

You could also read the input using `wscanf()`. The only difference is that you must specify the format string to be a wide character string:

```
value_count = wscanf(L"%f %d %d %[abcdefghijklmnopqrstuvwxyz] %*1d %s%n",
                              &fp1, &i , &j, word1, word2, &byte_count);
```

For this statement to compile, you need an `#include` directive for the `<wchar.h>` header file. The `L` prefix specifies in the first argument the string constant to be a wide character string, so it will occupy twice as much memory as a regular string. You would probably only use `wscanf()` if you were using wide character strings in your program, and in this case you would also be storing the strings read as wide character strings. Here's a fragment that illustrates how you could read strings as wide character strings:

```
wchar_t wword1[SIZE] = L" ";
wchar_t wword2[SIZE] = L" ";
value_count = wscanf(L"%l[abcdefghijklmnopqrstuvwxyz] %*1d %ls%n",
                              wword1, wword2, &byte_count);
printf("\nwword1 = %ls   wword2 = %ls\n", wword1, wword2);
```

Here you have two arrays that store elements of type `wchar_t` to hold the strings. The type specifiers in the format string for the two strings have `l` (lowercase L) as the prefix so the input is read as wide character strings. If you enter **ready2go**, then `ready` and `go` will be stored in `wword1` and `wword2` as wide character strings and the `2` between them will be discarded, as in the example. To output the strings, the `printf()` function uses `%ls` as the format specification, because the function needs to know they are wide character strings. If you were to use `%s`, the output would be incorrect. You could try it to see what you get.

Characters in the Input Format String

You can include a sequence of one or more characters that isn't a format conversion specifier within the input format string. If you do this, you're indicating that you expect the same characters to appear in the input and that the `scanf()` function should read them but not store them. These have to be matched exactly, character for character, by the data in the input stream. Any variation will terminate the input scanning process in `scanf()`.

TRY IT OUT: CHARACTERS IN THE INPUT FORMAT STRING

You can illustrate the effect of including characters in the input format string with the following example:

```
/* Program 10.2 Characters in the format control string */
#include <stdio.h>

int main(void)
{
  int i = 0;
  int j = 0;
  int value_count = 0;
  float fp1 = 0.0;

  printf("Input:\n");
  value_count = scanf("fp1 = %f i = %d %d", &fp1, &i , &j);

  printf("\nOutput:\n");
  printf("\nCount of values read = %d", value_count);
  printf("\nfp1 = %f\ti = %d\tj = %d\n", fp1, i, j);
  return 0;
}
```

Here's an example of the output:

```
Input:
fp1 = 3.14159 i = 7 8

Output:

Count of values read = 3
fp1 = 3.141590  i = 7   j = 8
```

How It Works

It doesn't matter whether the blanks before and after the = are included in the input—they're whitespace characters and are therefore ignored. The important thing is to include the same characters that appear in the format control string in the correct sequence and at the correct place in the input. Try an input line in which this isn't the case:

```
Input:
fp1 = 3.14159 i = 7 j = 8

Output:

Count of values read = 2
fp1 = 3.141590  i = 7   j = 0
```

Now only two values are read. This is because the character `j` in the input stops processing immediately, and no value is received by the variable `j`. The input processing of characters by `scanf()` is also case sensitive. If you input `Fp1=` instead of `fp1=`, no values will be processed at all, because the mismatch with the capital `F` will stop scanning before any values are entered.

Variations on Floating-Point Input

When you're reading floating-point values formatted using `scanf()`, you not only have a choice of specification that you use, but also you can enter the values in a variety of forms. You can see this with a simple example.

TRY IT OUT: FLOATING-POINT INPUT

With this example you can try various forms of specifier and various ways in which you can enter the input values.

```
/* Program 10.3 Floating-Point Input */
#include <stdio.h>

int main(void)
{
  float fp1 = 0.0f;
  float fp2 = 0.0f;
  float fp3 = 0.0f;
  int value_count = 0;

  printf("Input:\n");
  value_count = scanf("%f %f %f", &fp1, &fp2, &fp3);

  printf("\nOutput:\n");
  printf("Return value = %d", value_count);
  printf("\nfp1 = %f  fp2 = %f  fp3 = %f\n", fp1, fp2, fp3);
  return 0;
}
```

Here's an example of output from this program with the same input value written three different ways:

```
Input:
3.14.314E1.0314e+02

Output:
Return value = 3
fp1 = 3.140000  fp2 = 3.140000  fp3 = 3.140000
```

How It Works

This example demonstrates three different ways of entering the same value. The first way is a straightforward decimal value, the second has an exponent value defined by the `E1` that indicates that the value is to be multiplied by 10, and the third has an exponent value of `e+02` and therefore is to be multiplied by 100. As you can see, when you're reading a floating-point value with the `"%f"` specification, you have the option of whether to include an exponent. If you do include an exponent, you can define it beginning with either an `e` or an `E`. You also have the option to include a sign for the exponent value, `+` or `-`, and, of course, the value can be signed too. There are countless variations possible here.

You could try changing the `scanf()` statement to the following:

```
value_count = scanf("%e %g %f", &fp1, &fp2, &fp3);
```

Here's the output with this statement in the program:

```
Input:
3.14.314E1.0314e+02

Output:
Return value = 3
fp1 = 3.140000  fp2 = 3.140000  fp3 = 3.140000
```

Clearly all three format specifications work equally well with the various input forms. The variation between these is only when you use them for output with the `printf()` function.

I recommend that you experiment with the various possibilities here. In particular, try experimenting with floating-point numbers and the field-width specifiers for reading integers.

Reading Hexadecimal and Octal Values

As you saw earlier, you can read hexadecimal values from the input stream using the format specifier `%x`. For octal values you use `%o`. These are very straightforward, but let's see them working in an example.

TRY IT OUT: READING HEXADECIMAL AND OCTAL VALUES

Try the following example:

```
/* Program 10.4 Reading hexadecimal and octal values */
#include <stdio.h>

int main(void)
{
  int i = 0;
```

```
int j = 0;
  int k = 0;
  int n = 0;

  printf("Input:\n");
  n = scanf(" %d %x %o", &i , &j, &k );

  printf("\nOutput:\n");
  printf("%d values read.", n);
  printf("\ni = %d   j = %d   k = %d\n", i, j, k );
  return 0;
}
```

Here's some sample output:

```
Input:
12 12 12

Output:
3 values read.
i = 12   j = 18   k = 10
```

How It Works

You read the three values entered as 12. The first is read with a decimal format specifier `%d`, the second with a hexadecimal format specifier `%x`, and the third with an octal format specifier `%o`. The output shows that 12 in hexadecimal is 18 in decimal notation, whereas 12 in octal is 10 in decimal notation.

Hexadecimal data entry can be useful when you want to enter bit patterns (sequences of 1s and 0s), as they're easier to specify in hexadecimal than in decimal. Each hexadecimal digit corresponds to 4 bits, so you can specify a 16-bit word as four hexadecimal digits. Octal is hardly ever used, and it appears here mainly for historical reasons.

Note the following example of output:

```
Input:
18 18 18

Output:
3 values read.
i = 18   j = 24   k = 1
```

Here, the first two values are read correctly as 18, as a hexadecimal value is indeed 24 in decimal notation. However, the third value is read as 1. This is because 8 isn't a legal octal digit. Octal digits are 0 to 7.

You can enter hexadecimal values using `A` to `F`, or `a` to `f`, or even a mixture if you're so inclined. Here's another example of output:

```
Input:
12 aA 17

Output:
3 values read.
i = 12   j = 170   k = 15
```

The value aA is 10 × 16 + 10, which is 170 as a decimal value. The octal value 17 is 1 × 8 + 7, which is 15 as a decimal value.

There's no difference between using "%x" and "%X" with scanf(), but they'll have a different effect when you use them with printf() for output. You can demonstrate this by changing the last printf() statement to the following:

```
printf("\ni = %x   j = %X   k = %d\n", i, j, k );
```

This now outputs the first two values in hexadecimal notation. You can get the following output with the input shown:

```
Input:
26 AE 77

Output:
3 values read.
i = 1a   j = AE   k = 63
```

So "%x" produces hexadecimal output using hexadecimal digits a to e, and "%X" produces output using hexadecimal digits A to E.

Reading Characters Using scanf()

You tried reading strings in the first example, but there are more possibilities. You know that there are three format specifiers for reading one or more single-byte characters. You can read a single character and store it as type char using the format specifier %c and as type wchar_t using %lc. For a string of characters, you use either the specifier %s or the specifier %[], or if you are storing the input as wide characters, %ls or %l[], where the prefix to the conversion specification is lowercase *L*. In this case, the string is stored as a null-terminated string with '\0' as the last character. With %[] or %l[] format specification, the string to be read must include only the characters that appear between the square brackets, or if the first character between the square brackets is ^, the string must contain only characters that are *not* among those following the ^ characters. Thus, %[aeiou] will read a string that consists only of vowels. The first character that isn't a vowel will signal the end of the string. The specification %[^aeiou] reads a string that contains any character that isn't a vowel. The first vowel will signal the end of the string.

Note that one interesting aspect of the %[] specification is it enables you to read a string containing spaces, something that the %s specification can't do. You just need to include a space as one of the characters between the square brackets.

TRY IT OUT: READING CHARACTERS AND STRINGS

You can see these character-reading capabilities in operation with the following example:

```
/* Program 10.5 Reading characters with scanf() */
#include <stdio.h>

int main(void)
{
  char initial = ' ';
  char name[80] = { 0 };
```

```
 char age[4] = { 0 };
  printf("Enter your first initial: ");
  scanf("%c", &initial );
  printf("Enter your first name: " );
  scanf("%s", name );

  if(initial  != name[0])
    printf("\n%s,you got your initial wrong.", name);
  else
    printf("\nHi, %s. Your initial is correct. Well done!", name );
  printf("\nEnter your full name and your age separated by a comma:\n" );
  scanf("%[^,] , %[0123456789]", name, age );
  printf("\nYour name is %s and you are %s years old\n", name, age );
  return 0;
}
```

Here's some output from this program:

```
Enter your first initial: I
Enter your first name: Ivor

Hi, Ivor. Your initial is correct. Well done!
Enter your full name and your age separated by a comma:
Ivor Horton    ,  99

Your name is
Ivor Horton     and you are 99 years old
```

How It Works

This program first expects you to enter your first initial and then your first name. It checks that the first letter of your name is the same as the initial you entered. This works in a straightforward way, as you can see from the output.

Next, you're asked to enter your full name followed by your age, separated by a comma. The read operation is carried out by the following statement:

```
scanf("%[^,] , %[0123456789]", name, age );
```

I deliberately spaced out the input data so you could see that the first input specification, `%[^,]`, reads any character as part of the string that isn't a comma, including spaces. Hence the extra spaces following the name in the last line of output. You then have a comma in the control string that will cause `scanf()` to read the comma (or several commas in succession) in the input and not store it. The input for `age` is read as a string with the specifier `%[0123456789]`. This will read any sequence of consecutive digits as a string.

Note that the comma in the input string is essential for the input to be read properly. If you leave it out, `scanf()` will attempt to read the comma as part of the input for age. Because a comma is evidently not a digit, this will stop input for `age` so it will just consist of an empty string.

If you try entering a space and then your initial as the first input, the program will treat the blank as the value for `initial` and the single character you entered as your `name`. With the way the control string is defined, the first character that you enter when using the `%c` specifier is taken to be the character, whatever it is. If you don't want a space to be accepted as the initial, you can fix this by writing the input statement as follows:

```
scanf(" %c", &initial );
```

Now the first character in the control string is a space, so `scanf()` will read and ignore any number of spaces and read the first character that isn't a space into `initial`.

Pitfalls with scanf()

There are two very common mistakes people make when using `scanf()` that you should keep in mind:

- Don't forget that the arguments *must* be pointers. Perhaps the most common error is to forget the ampersand (`&`) when specifying single variables as arguments to `scanf()`, particularly because you don't need it with `printf()`. Of course, the `&` isn't necessary if the argument is an array name or a pointer variable.
- When reading a string, remember to ensure that there's enough space for the string to be read in, *plus* the terminating `'\0'`. If you don't do this, you'll overwrite something in memory, possibly even some of your program code.

String Input from the Keyboard

As you've seen, the `gets()` function in `<stdio.h>` will read a complete line of text as a string. The prototype of the function is as follows:

```
char *gets(char *str);
```

This function reads successive characters into the memory pointed to by `str` until you press the Enter key. It appends the terminating null, `'\0'`, in place of the newline character that is read when you press the Enter key. The return value is identical to the argument, which is the address where the string has been stored. The following example provides a reminder of how it works.

TRY IT OUT: READING A STRING WITH GETS()

Here's a simple program using `gets()`:

```
/* Program 10.6 Reading a string with gets() */
#include <stdio.h>

int main(void)
{
  char initial[2] = {0};
  char name[80] = {0};

  printf("Enter your first initial:  ");
  gets(initial);
  printf("Enter your name:  " );
  gets(name);
  if(initial[0] != name[0])
    printf("\n%s,you got your initial wrong.\n", name);
  else
    printf("\nHi, %s. Your initial is correct. Well done!\n", name);
  return 0;
}
```

Here's some output from this program:

```
Enter your first initial:  M
Enter your name:  Mephistopheles

Hi, Mephistopheles. Your initial is correct. Well done!
```

How It Works

You read the initial and the name as strings using `gets()`. The function is very easy to use because there's no format specification involved. Because `gets()` will read characters until you press Enter, you can now enter your full name if you wish.

Of course, the downside to using `gets()` is that you have no control over how many characters are stored. This implies that you must be sure to create an array to receive the data with sufficient space for the maximum length string that might be entered. Where you want to be sure that your array length will not be exceeded, you have the option of using the `fgets()` function. You could replace the statements that manage the input with the following:

```
printf("Enter your first initial: ");
fgets(initial, sizeof(initial), stdin);     /* Read 1 character max       */
fflush(stdin);                              /* Flush the newline          */

printf("Enter your name: " );
fgets(name, sizeof(name), stdin);           /* Read max name-1 characters */
size_t length = strlen(name);
name[length-1] = name[length];              /* Overwrite the newline      */
```

The `fgets()` function will read up to one less than the number of characters specified by the second argument and append the terminating `'\0'`. This ensures that the length of the array you pass as the first argument is not exceeded. You need to remember that the `fgets()` function stores a newline character in the input string corresponding to the Enter key being pressed, whereas the `gets()` function does not. With the first read operation, the newline will be left in the input buffer so the call to `fflush()` will flush `stdin` and remove it. Without this, the newline would be read as the input for `name`. The last character before the null in `name` will be the newline character, so copying the terminating null one position back will overwrite it.

For string input, using `gets()` or `fgets()` is usually the preferred approach unless you want to control the content of the string, in which case you can use `%[]`. The `%[]` specification is more convenient to use when the nonstandard `%[a-z]` form is supported, but remember, because this is nonstandard, your code is no longer as portable as it is if you use the standard form for reading a string of lowercase letters, `%[abcdefghijklmnopqrstuvwxyz]`.

Unformatted Input from the Keyboard

The `getchar()`function reads one character at a time from `stdin`. The `getchar()` function is defined in `<stdio.h>`, and its general syntax is as follows:

```
int getchar(void);
```

The `getchar()` function requires no arguments, and it returns the character read from the input stream. Note that this character is returned as `int`, and the character is displayed on the screen as it is entered from the keyboard.

With many implementations of C, the nonstandard header file `<conio.h>` is often included. This provides additional functions for character input and output. One of the most useful of these is `getch()`, which reads a character from the keyboard without displaying it on the screen. This is

particularly useful when you need to prevent others from being able to see what's being keyed in—for example, when a password is being entered.

The standard header <stdio.h> also declares the ungetc() function that enables you to put a character that you have just read back into an input stream. The function requires two arguments: the first is the character to be pushed back onto the stream, and the second is the identifier for the stream, which would be stdin for the standard input stream. The ungetc() returns a value of type int that corresponds to the character pushed back onto the stream, or a special character, EOF (end-of-file), if the operation fails.

In principle you can push a succession of characters back into an input stream but only one character is guaranteed. As I noted, a failure to push a character back onto a stream will be indicated by EOF being returned by the function, so you should check for this if you are attempting to return several characters to a stream.

The ungetc() function is useful when you are reading input character by character and don't know how many characters make up a data unit. You might be reading an integer value, for example, but don't know how many digits there are. In this situation the ungetc() function makes it possible for you to read a succession of characters using getchar(), and when you find you have read a character that is not a digit, you can return it to the stream. Here's a function that ignores spaces and tabs from the standard input stream using the getchar() and ungetc() functions:

```
void eatspaces(void)
{
  char ch = 0;
  while(isspace(ch = getchar()));  /* Read as long as there are spaces */
  ungetc(ch, stdin);               /* Put back the nonspace character */
}
```

The isspace() function that is declared in the <ctype.h> header file returns true when the argument is a space character. The while loop continues to read characters as long as they are spaces or tabs, storing each character in ch. The first nonspace character that is read will end the loop, and the character will be left in ch. The call to ungetc() returns the nonblank character back to the stream for future processing.

Let's try out the getchar() and ungetc() functions in a working example.

TRY IT OUT: READING AND UNREADING CHARACTERS

This example will assume the input from the keyboard consists of some arbitrary sequence of integers and names:

```
/* Program 10.7 Reading and unreading characters */
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>
#include <string.h>

const size_t LENGTH = 50;

/* Function prototypes */
void eatspaces(void);
bool getinteger(int *n);
char *getname(char *name, size_t length);
bool isnewline(void);

int main(void)
```

```
{
  int number;
  char name[LENGTH];
  printf("Enter a sequence of integers and alphabetic names:\n");
  while(!isnewline())
    if(getinteger(&number))
      printf("\nInteger value:%8d", number);
    else if(strlen(getname(name, LENGTH)) > 0)
      printf("\nName: %s", name);
    else
    {
      printf("\nInvalid input.");
      return 1;
    }
  return 0;
}

/* Function to check for newline */
bool isnewline(void)
{
  char ch = 0;
  if((ch = getchar()) == '\n')
    return true;

  ungetc(ch, stdin);
  return false;
}

/* Function to ignore spaces from standard input */
void eatspaces(void)
{
  char ch = 0;
  while(isspace(ch = getchar()));
  ungetc(ch, stdin);
}

/* Function to read an integer from standard input */
bool getinteger(int *n)
{
  eatspaces();
  int value = 0;
  int sign = 1;
  char ch = 0;

  /* Check first character */
  if((ch=getchar()) == '-')            /* should be minus */
    sign = -1;
  else if(isdigit(ch))                 /* ...or a digit   */
   value = 10*value + (ch - '0');
  else  if(ch != '+')                  /* ...or plus      */
  {
    ungetc(ch, stdin);
    return false;                      /* Not an integer  */
  }
```

```
  /* Find more digits */
  while(isdigit(ch = getchar()))
    value = 10*value + (ch - '0');

  /* Push back first nondigit character */
  ungetc(ch,stdin);
  *n = value*sign;
  return true;
}

/* Function to read an alphabetic name from input */
char *getname(char *name, size_t length)
{

  eatspaces();                          /* Remove leading spaces */
  size_t count = 0;
  char ch = 0;
  while(isalpha(ch=getchar()))          /* As long as there are letters */
  {
    name[count++] = ch;                 /* store them in name           */
    if(count == length-1)
      break;
  }

  name[count] = '\0';                   /* Append string terminator     */
  if(count < length-1)
    ungetc(ch, stdin);                  /* Return nonletter to stream  */
  return name;
}
```

Here's an example of output from the program:

```
Enter a sequence of integers and alphabetic names:
12              Jack Jim  234 Jo Janet 99  88

Integer value:      12
Name: Jack
Name: Jim
Integer value:     234
Name: Jo
Name: Janet
Integer value:      99
Integer value:      88
```

Here's another sample:

```
Enter a sequence of integers and alphabetic names:
Jim     Jo Will Bert

Name: Jim
Name: Jo
Name: Will
Name: Bert
```

How It Works

There are four functions using the `getchar()` and `ungetc()` functions to read from `stdin`. You saw the `eatspaces()` function in the previous section. The `isnewline()` function just reads a character from the keyboard and returns `true` if it is a newline character. This function is used to control when input ends in `main()`.

The `getinteger()` function reads an integer of arbitrary length from the keyboard that is optionally preceded by a sign. The first step is to remove leading spaces by calling the `eatspaces()` function. After checking for a sign or the first digit, the function continues to read digits from the keyboard in a loop:

```
while(isdigit(ch = getchar()))
  value = 10*value + (ch - '0');
```

The digits are read from left to right, so the latest digit is the low-order digit in the number. The digit value is obtained by subtracting the code value for the 0 digit from the code value for the current digit. This works because the code values for digits are in ascending sequence. To insert the digit, you multiply the current accumulated value by 10 and add the new digit value. Of course, storing the result as type `int` is a constraint. You could implement the function to store the value as type `long long` to accommodate a wider range of values. You could also include code to check for how large the number is getting and to output an error message if it cannot be stored as type `int`.

The first character read that is not a digit ends the loop, and this character is returned to the stream so that it can be read again.

The `getname()` function reads an alphabetic name from the keyboard. The arguments are an array in which the name is to be stored, and the length of the array so the function can ensure the capacity is not exceeded. The function returns the address of the first byte of the string as a convenience to the calling program. The process is, in principle, the same as the `getinteger()` function. The function continues to read characters in a loop as long as they are alphabetic characters:

```
while(isalpha(ch=getchar()))         /* As long as there are letters */
{
  name[count++] = ch;                /* store them in name           */
  if(count == length-1)
    break;
}
```

The `count` variable tracks the number of characters stored in the `name` array, and when only one element is still free, the loop ends. After the loop, the code appends a `'\0'` to terminate the string. Of course, the last character read will have been alphabetic and therefore stored in the array if the value of `count` reaches `length-1`. You therefore only restore the last character back to the stream by calling `ungetc()` when this is not the case.

The `main()` function reads an arbitrary sequence of names and integers in a loop:

```
while(!isnewline())
  if(getinteger(&number))
    printf("\nInteger value:%8d", number);
  else if(strlen(getname(name, LENGTH)) > 0)
    printf("\nName: %s", name);
  else
  {
    printf("\nInvalid input.");
    return 1;
  }
```

The loop continues as long as the current character is not a newline signaling the end of the current line. The program expects to read either an integer or a name on each loop iteration. The loop first tries to read an integer by calling the `getinteger()` function. This function returns `false` if an integer is not found, in which case the `getname()` function is called to read a name. If no name is found, the input is neither a name nor an integer, so the program ends after outputting a message.

Output to the Screen

Writing data to the command line on the screen is much easier than reading input from the keyboard. You know what data you're writing, whereas with input you have all the vagaries of possible incorrect entry of the data. The primary function for formatted output to the `stdout` stream is `printf()`.

Fortunately—or unfortunately, depending how you view the chore of getting familiar with this stuff—`printf()` provides myriad possible variations for the output you can obtain, much more than the scope of the format specifiers associated with `scanf()`.

Formatted Output to the Screen Using printf()

The `printf()` function is defined in the header file `<stdio.h>`, and its general form is the following:

```
int printf(char *format, ...);
```

The first parameter is the format control string. The argument for this parameter is usually passed to the function as an explicit string constant, as you've seen in all the examples, but it can be a pointer to a string that you specify elsewhere. The optional arguments to the function are the values to be output in sequence, and they must correspond in number and type with the format conversion specifiers that appear in the string that is passed as the first argument. Of course, as you've also seen in earlier examples, if the output is simply the text that appears in the control string, there are no additional arguments after the first. But where there are argument values to be output, there must be *at least* as many arguments as there are format specifiers. If not, the results are unpredictable. If there are *more* arguments than specifiers, the excess is ignored. This is because the function uses the format string as the determinant of how many arguments follow and what type they have.

■ **Note** The fact that the format string alone determines how the data is interpreted is the reason why you get the wrong result with a `%d` specifier combined with a `long long` argument.

The `<stdio.h>` header also declares the `wprintf()` function. Analogous to the situation you saw with `scanf()`, the `wprintf()` function works in exactly the same way as `printf()` except that it expects

the first argument to be a wide character string. The format specifications are exactly the same for both functions.

The format conversion specifiers for `printf()` and `wprintf()` are a little more complicated than those you use for input with `scanf()` and `wscanf()`. The general form of an output format specifier is shown in Figure 10-3.

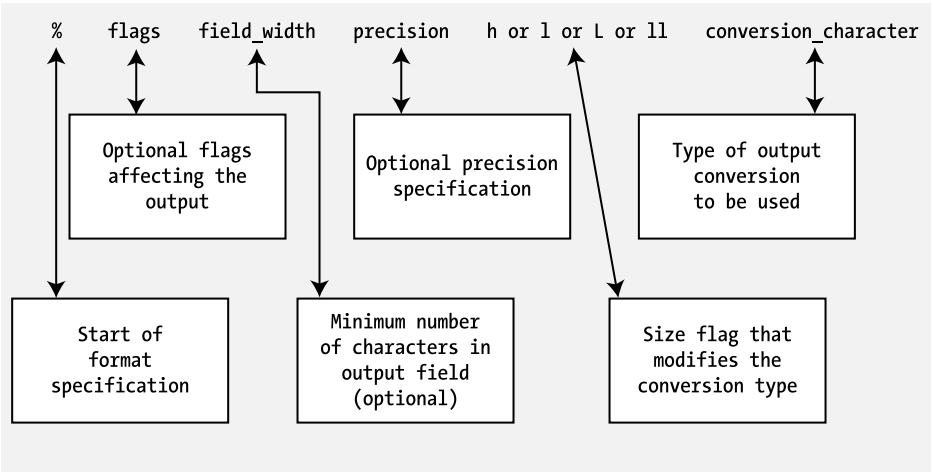


Figure 10-3. *Format specifications for the `printf()` and `wprintf()` functions*

You've seen most of the details before, but let's take a quick pass through the elements of this general format specifier:

- The `%` sign indicates the start of the specifier, as it does for output.
- The optional flag characters are `+`, `-`, `#`, and space. These affect the output, as shown in Table 10-4.

Table 10-4. *Effects of the Optional Flag Characters in an Output Specification*

Character	Use
`+`	Ensures that, for signed output values, there's always a sign preceding the output value—either a plus or a minus sign. By default, only negative values have a sign.
`-`	Specifies that the output value is left-justified in the output field and padded with blanks on the right. The default positioning of the output is right-justified.
`0`	Prefixes a `field_width` value to specify that the value should be padded with zeros to fill out the field width to the left.
`#`	Specifies that `0` is to precede an octal output value, `0x`; or `0X` is to precede a hexadecimal output value; or a floating-point output value will contain a decimal point. For the `g` or `G` floating-point conversion characters, trailing zeros will also be omitted.
space	Specifies that positive or zero output values are preceded by a space rather than a plus sign.

- The optional `field_width` specifies the minimum number of characters for the output value. If the value requires more characters, the field is simply expanded. If it requires less than the minimum specified, it is padded with blanks, unless the field width is specified with a leading zero, as in `09`, for example, where it would be filled on the left with zeros.
- The precision specifier is also optional and is generally used with floating-point output values. A specifier of `.n` indicates that `n` decimal places are to be output. If the value to be output has more than `n` significant digits, it's rounded or truncated.
- You prefix the appropriate type conversion character with the `h`, `l` (lowercase letter *L*), `ll`, or `L` modifier to specify that the output conversion is being applied to `short`, `long`, `long long`, or `long double` values, respectively. The `l` modifier applied to the `c` type specification specifies that the character is to be stored as type `wchar_t`.
- The conversion character that you use defines how the output is to be converted for a particular type of value. Conversion characters are defined in Table 10-5.

Table 10-5. *Conversion Characters in an Output Specification*

Conversion Character	Output Produced
Applicable to integers	
`d`	Signed decimal integer value
`o`	Unsigned octal integer value
`u`	Unsigned decimal integer value
`x`	Unsigned hexadecimal integer value with lowercase hexadecimal digits `a`, `b`, `c`, `d`, `e`, `f`
`X`	As `x` but with uppercase hexadecimal digits `A`, `B`, `C`, `D`, `E`, `F`
Applicable to floating-point	
`f`	Signed decimal value
`e`	Signed decimal value with exponent
`E`	As `e` but with `E` for exponent instead of `e`
`g`	As `e` or `f` depending on size of value and precision
`G`	As `g` but with `E` for exponent values
Applicable to characters	
`c`	Single character
`s`	All characters until `'\0'` is reached or `precision` characters have been output

Believe it or not, this set of output options includes only the most important ones. If you consult the documentation accompanying your compiler, you'll find a few more.

Escape Sequences

You can include whitespace characters in the format control string for `printf()` and `wprintf()`. The characters that are referred to as whitespace are the newline, carriage return, and form-feed characters; blank (a space); and tab. Some of these are represented by escape sequences that begin with `\`. Table 10-6 shows the most common escape sequences.

You use the escape sequence `\\` in format control strings when you want to output the backslash character `\`. If this weren't the case, it would be impossible to output a backslash, because it would always be assumed that a backslash was the start of an escape sequence. To write a `%` character to `stdout`, use `%%`. You can't use `%` by itself as this would be interpreted as the start of a format specification.

Table 10-6. *Common Escape Sequences*

Escape Sequence	Description
\a	Bell sound (a beep on your computer not used much these days)
\b	Backspace
\f	Form-feed or page eject
\n	Newline
\r	Carriage return (for printers) or move to the beginning of the current line for output to the screen
\t	Horizontal tab

■ **Note** Of course, you can use escape sequences within any string, not just in the context of the format string for the `printf()` function.

Integer Output

Let's take a look at some of the variations that you haven't made much use of so far. Those with field width and precision specifiers are probably the most interesting.

TRY IT OUT: OUTPUTTING INTEGERS

Let's try a sample of integer output formats first:

```
/* Program 10.8 Integer output variations */
#include <stdio.h>

int main(void)
{
  int i = 15;
  int j = 345;
  int k = 4567;
  long li = 56789L;
  long lj = 678912L;
  long lk = 23456789L;

  printf("\ni = %d   j = %d    k = %d   i = %6.3d   j = %6.3d   k = %6.3d\n",
                                                    i ,j, k, i, j, k);
  printf("\ni = %-d   j = %+d   k = %-d   i = %-6.3d   j = %-6.3d   k ="
                                       " %-6.3d\n",i ,j, k, i, j, k);
  printf("\nli = %d   lj = %d   lk = %d\n", li, lj, lk);
  printf("\nli = %ld   lj = %ld   lk = %ld\n", li, lj, lk);
  return 0;
}
```

When you execute this example, you should see something like this:

```
i = 15   j = 345    k = 4567   i =    015   j =    345   k =   4567

i = 15   j = +345   k = 4567   i = 015      j = 345      k = 4567

li = -8747   lj = 23552   lk = -5099

li = 56789   lj = 678912   lk = 23456789
```

How It Works

This example illustrates a miscellany of options for integer output. You can see the effects of the - flag by comparing the first two lines produced by these statements:

```
printf("\ni = %d   j = %d    k = %d   i = %6.3d   j = %6.3d   k = %6.3d\n",
                                                    i ,j, k, i, j, k);
printf("\ni = %-d   j = %+d   k = %-d   i = %-6.3d   j = %-6.3d   k ="
                                      " %-6.3d\n",i ,j, k, i, j, k);
```

The - flag causes the output to be left-justified. The effect of the field width specifier is also apparent from the spacing of the last three outputs in each group of six. Note that the default width provides just enough output positions to accommodate the number of digits to be output, so the - flag has no effect.

You get a leading plus in the output of `j` on the second line because of the flag modifier. You can use more than one flag modifier if you want. With the second output of the value of `i`, you have a leading 0 inserted due to the minimum precision being specified as 3. You could also have obtained leading zeroes by preceding the minimum width value with a 0 in the format specification.

The third output line is produced by the following statement:

```
printf("\nli = %d   lj = %d   lk = %d\n", li, lj, lk);
```

Here, you can see that failure to insert the `l` (lowercase letter *L*) modifier when outputting integers of type `long` results in apparent garbage, because the output value is assumed to be a 2-byte integer. Of course, if your system implements type `int` as a 4-byte integer, the values will be correct here. The problem arises only if `long` and `int` are differentiated. You should get a warning from your compiler if there is a mismatch. The same problem can arise when you use the wrong type conversion when outputting values of type `long long`.

You get the correct values from this statement:

```
printf("\nli = %ld   lj = %ld   lk = %ld\n", li, lj, lk);
```

It's unwise to specify inadequate values for the width and the precision of the values to be displayed. Weird and wonderful results may be produced if you do. Try experimenting with this example to see just how much variation you can get.

TRY IT OUT: VARIATIONS ON A SINGLE INTEGER

You'll try one more integer example to run the gamut of possibilities with a single integer value:

```
/* Program 10.9 Variations on a single integer */
#include <stdio.h>

int main(void)
{
  int k = 678;

  printf("%%d   %%o   %%x   %%X");  /* Display format as heading */
  printf("\n%d  %o  %x  %X", k, k, k, k );      /* Display values */

  /* Display format as heading then display the values */
  printf("\n\n%%8d       %%-8d      %%+8d      %%O8d      %%-+8d");
  printf("\n%8d  %-8d  %-+8d  %O8d  %-+8d\n", k, k, k, k, k );
  return 0;
}
```

How It Works

This program may look a little confusing at first because the first of each pair of `printf()` statements displays the format used to output the number appearing immediately below. The `%%` specifier simply outputs the `%` character.

When you execute this example, you should get something like this:

```
%d   %o   %x   %X
678  1246  2a6  2A6

%8d       %-8d      %+8d      %O8d      %-+8d
678       678       +678      00000678  +678
```

The first row of output values is produced by this statement:

```
printf("\n%d  %o  %x  %X", k, k, k, k );      /* Display values */
```

The outputs are decimal, octal, and two varieties of hexadecimal for the value 678, with the default width specification. The corresponding format appears above each value in the output.

The next row of output values is produced by this:

```
printf("\n%8d  %-8d  %-+8d  %O8d  %-+8d\n", k, k, k, k, k );
```

This statement includes a variety of flag settings with a width specification of `8`. The first is the default right-justification in the field. The second is left-justified because of the `-` flag. The third has a sign because of the `+` flag. The fourth has leading zeroes because the width is specified as `O8` instead of `8`, and it also has a sign because of the `+` flag. The last output value uses a specifier with all the trimmings, `%-+8d`, so the output is left-justified in the field and also has a leading sign.

Tip When you're outputting multiple rows of values on the screen, using a width specification—possibly with tabs—will enable you to line them up in columns.

Outputting Floating-Point Values

If plowing through the integer options hasn't started you nodding off, then take a quick look at the floating-point output options through working examples.

TRY IT OUT: OUTPUTTING FLOATING-POINT VALUES

Look at the following example:

```
/* Program 10.10 Outputting floating-point values */
 #include <stdio.h>

int main(void)
{
  float fp1 = 345.678f;
  float fp2 = 1.234E6f;
  double fp3 = 234567898.0;
  double fp4 = 11.22334455e-6;

  printf("\n%f  %+f  %-10.4f  %6.4f\n", fp1, fp2, fp1, fp2);
  printf("\n%e  %+E\n", fp1, fp2);
  printf("\n%f  %g  %#+f  %8.4f  %10.4g\n", fp3,fp3, fp3, fp3, fp4);
  return 0;
}
```

How It Works

With my compiler, I get this output:

```
345.678009      +1234000.000000 345.6780        1234000.0000

3.456780e+002   +1.234000E+006

234567898.000000        2.34568e+008    +234567898.000000       234567898.0000

1.122e-005
```

It's possible that you may not get exactly the same output, but it should be close. Most of the output is a straightforward demonstration of the effects of the format conversion specifiers that I've discussed, but a few points are noteworthy.

The value of the first output for `fp1` differs slightly from the value that you assigned to the variable. This is typical of the kind of small difference that can creep in when floating-point numbers are converted from decimal to binary. With fractional decimal values, there isn't always an exact equivalent in binary floating-point.

In the output generated by the statement

```
printf("\n%f  %+f  %-10.4f  %6.4f\n", fp1, fp2, fp1, fp2);
```

the second output value for `fp1` shows how the number of decimal places after the point can be constrained. The output in this case is left-justified in the field. The second output of `fp2` has a field width specified that is too small for the number of decimal places required and is therefore overridden.

The second `printf()` statement is as follows:

```
printf("\n%e  %+E\n", fp1, fp2);
```

This outputs the same values in floating-point format with an exponent. Whether you get an uppercase `E` or a lowercase `e` for the exponent indicator depends on how you write the format specifier.

In the last line, you can see how the `g`-specified output of `fp3` has been rounded up compared to the `f` specified output.

Note There are a huge number of possible variations for the output obtainable with `printf()`. It would be very educational for you to play around with the options, trying various ways of outputting the same information.

Character Output

Now that you've looked at the various possibilities for outputting numbers, let's have a look at outputting characters. There are basically four flavors of output specifications you can use with `printf()` and `wprintf()` for character data: `%c` and `%s` for single characters and strings, and `%lc` and `%ls` for single wide characters and wide character strings, respectively. You have seen how to use `%s` and `%ls`, so let's just try out single character output in an example.

TRY IT OUT: OUTPUTTING CHARACTER DATA

This example outputs all the printable characters, then all the lower and uppercase letters:

```
/* Program 10.11 Outputting character data */
#include <stdio.h>
#include <limits.h>
#include <wchar.h>
#include <ctype.h>
#include <wctype.h>

int main(void)
{
  int count = 0;
  char ch = 0;
  printf("\nThe printable characters are the following:\n");

  /* Iterate over all values of type char */
  for(int code = 0 ; code <= CHAR_MAX ; code++)
  {
    ch = (char)code;
    if(isprint(ch))
    {
      if(count++ % 32 == 0)
        printf("\n");
      printf("%c", ch);
    }
  }
```

```
  /* Use wprintf() to output wide characters */
  count = 0;
  wchar_t wch = 0;
  wprintf(
      L"\n\nThe alphabetic characters and their codes are the following:\n");

  /* Iterate over the lowercase wide character letters */
  for(wchar_t wch = L'a' ; wch <= L'z' ; wch++)
  {
    if(count++ % 3 == 0)
      wprintf(L"\n");

    wprintf(L" %lc    %#x    %lc    %#x", wch, (long)wch, towupper(wch),
                                                    (long)towupper(wch));
  }
  return 0;
}
```

The output from this program is the following:

```
The printable characters are the following:

 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~

The alphabetic characters and their codes are the following:

 a    0x61    A    0x41 b    0x62    B    0x42 c    0x63    C    0x43
 d    0x64    D    0x44 e    0x65    E    0x45 f    0x66    F    0x46
 g    0x67    G    0x47 h    0x68    H    0x48 i    0x69    I    0x49
 j    0x6a    J    0x4a k    0x6b    K    0x4b l    0x6c    L    0x4c
 m    0x6d    M    0x4d n    0x6e    N    0x4e o    0x6f    O    0x4f
 p    0x70    P    0x50 q    0x71    Q    0x51 r    0x72    R    0x52
 s    0x73    S    0x53 t    0x74    T    0x54 u    0x75    U    0x55
 v    0x76    V    0x56 w    0x77    W    0x57 x    0x78    X    0x58
 y    0x79    Y    0x59 z    0x7a    Z    0x5a
```

How It Works

The first block of output for the printable characters is generated in the `for` loop:

```
  for(int code = 0 ; code <= CHAR_MAX ; code++)
  {
    ch = (char)code;
    if(isprint(ch))
    {
      if(count++ % 32 == 0)
        printf("\n");
      printf("%c", ch);
    }
  }
```

First, note the type of the loop control variable, code. You might be tempted to use type char here but this would be a serious mistake, as the loop would run indefinitely. The reason for this is that the condition for ending the loop is checked after the value of code has been incremented. On the last good iteration, code has the value CHAR_MAX, which is the maximum value that can be stored as type char. If code is type char, when 1 is added to CHAR_MAX the result would be 0, so the loop would continue instead of ending as it should.

Within the loop you cast the value of code to type char and store the result in ch. The explicit cast is not required for the code to compile, as the compiler will insert the conversion, but it does indicate that it is intentional. You then use the isprint() function that is declared in the <ctype.h> header to test for a printable character. When isprint() returns true, you output the character using the %c format specification. You also arrange to output a newline character each time 32 characters have been output so that you don't have output just spilling arbitrarily from one line to the next.

The second loop uses the wprintf() function to output alphabetic wide characters and their character code values:

```
for(wchar_t wch = L'a' ; wch <= L'z' ; wch++)
{
  if(count++ % 3 == 0)
    wprintf(L"\n");

  wprintf(L" %lc    %#x    %lc    %#x", wch, (long)wch, towupper(wch),
                                                    (long)towupper(wch));
}
```

This time you can use a loop control variable wch of type wchar_t that iterates over the code values from L'a' to L'z'. You use essentially the same trick as you used in the previous loop to output three success groups of output on each line. The wprintf() outputs the character using the %lc specification and the code value as %ld. To output the code value you cast the value stored in wch to type long and use the %ld format specification to display it. You use the towupper() function that is declared in the <wctype.h> header to get the uppercase equivalent of wch.

Don't forget, the only difference between the wprintf() and printf() functions is that the former requires the first argument to be a wide character string. All of the last output operation could be done just as well with printf(); you would just need to remove the L prefix from the format string that is the first argument.

Other Output Functions

In addition to the string output capabilities of printf() and wprintf(), you have the puts() function that is also declared in <stdio.h>, and which complements the gets() function. The name of the function derives from its purpose: **put** string. The general form of puts() is as follows:

```
int puts(const char *string);
```

The puts() function accepts a pointer to a string as an argument and writes the string to the standard output stream, stdout. The string must be terminated by '\0'. The parameter to puts() is const so the function does not modify the string you pass to it. The function returns a negative integer value if any errors occur on output, and a nonnegative value otherwise. The puts() function is very useful for outputting single-line messages, for example:

```
puts("Is there no end to input and output?");
```

This will output the string that is passed as the argument and then move the cursor to the next line. The function `printf()` requires an explicit `'\n'` to be included at the end of the string to do the same thing.

■ **Note** The function `puts()` will process embedded `'\n'` characters in the string you pass as the argument to generate output on multiple lines.

Unformatted Output to the Screen

Also included in `<stdio.h>`, and complementing the function `getchar()`, is the function `putchar()`. This has the following general form:

```
int putchar(int c);
```

The `putchar()` function outputs a single character, `c`, to `stdout` and returns the character that was displayed. This allows you to output a message one character at a time, which can make your programs a bit bigger, but gives you control over whether or not you output particular characters. For example, you could simply write the following to output a string:

```
char string[] = "Beware the Jabberwock, \nmy son!";
puts(string);
```

Alternatively, you could write:

```
char string[] = " Beware the Jabberwock, \nmy son!";
int i = 0;
while( string[i] != '\0')
  if(string[i] != '\n')
    putchar(string[i++]);
```

The first fragment outputs the string over two lines, like this:

```
Beware the Jabberwock,
my son!
```

The second fragment skips newline characters in the string so the output will be the following:

```
Beware the Jabberwock, my son!
```

Your use of `putchar()` need not be as simple as this. With `putchar()` you could choose to output a selected sequence of characters from the middle of a string bounded by a given delimiter, or you could selectively transform the occurrences of certain characters within a string before output, converting tabs to spaces, for example.

Formatted Output to an Array

You can write formatted data to an array of elements of type `char` in memory using the `sprintf()` function that is declared in the `<stdio.h>` header file. This function has the prototype:

```
int sprintf(char *str, const char *format, . . .);
```

The function writes the data specified by the third and subsequent arguments formatted according to the format string second argument. This works identically to `printf()` except that the data is written to the string specified by the first argument to the function. The integer returned is a count of the number of characters written to `str`, excluding the terminating null character.

Here's a fragment illustrating how this works:

```
char result[20];                /* Output from sprintf */
int count = 4;
int nchars = sprintf(result, "A dog has %d legs.", count);
```

The effect of this is to write the value of `count` to `result` using the format string that is the second argument. The effect is that `result` will contain the string `"A dog has 4 legs."`. The variable `nchars` will contain the value 17, which is the same as the return value you would get from `strlen(result)` after the `sprintf()` call has executed. The `sprintf()` function will return a negative integer if an encoding error occurs during the operation.

One use for the `sprintf()` function is to create format strings programmatically. You'll see a simple example of this in Program12.8 in Chapter 12.

Formatted Input from an Array

The `sscanf()` function complements the `sprintf()` function because it enables you to read data under the control of a format string from an array of elements of type char. The prototype looks like this:

```
int sscanf(const char *str, const char *format, . . .);
```

Data will be read from `str` into variables that are specified by the third and subsequent arguments according to the format string `format`. The function returns a count of the number of items actually read, or `EOF` if a failure occurs before any data values are read and stored. The end of the string is recognized as the end-of-file condition, so reaching the end of the `str` string before any values are converted will cause `EOF` to be returned.

Here's a simple illustration of how the `sscanf()` function works:

```
char *source = "Fred 94";
char name[10];
int age = 0;
int items = sscanf(source, " %s %d", name, age);
```

The result of executing this fragment is that `name` will contain the string `"Fred"` and `age` will contain the value 94. The variable `items` will contain the value 2 because two items are read from `source`.

One use for the `sscanf()` function is to try various ways of reading the same data. You can always read an input line into an array as a string. You can then use `sscanf()` to reread the same input line from the array with different format strings as many times as you like.

Sending Output to the Printer

Writing to a printer is not part of standard C, but some C libraries do support it, so it's worth mentioning. To write output to the default printer, you can use a more generalized form of the `printf()` function called `fprintf()`. This function is designed to send formatted output to *any* stream, and more often to files on disk, but I'll stick to printing for now. For the purpose of printing, the general form for using `fprintf()` is as follows:

```
fprintf(stdprn, format_string, argument1, argument2, ..., argumentn);
```

The function will return a value of type `int` that is a count of the number of values that were sent to `stdprn`. With the exception of the first argument and the extra `f` in the function name, `fprintf()` looks exactly like `printf()`. And so it is. If you don't have `stdprn` defined with your compiler and library, you'll need to consult your documentation to see how to handle printing, but many C libraries do define `stdprn`. You can use the same format string with the same set of specifiers to output data to your printer in exactly the same way that you display results with `printf()`. However, there are a couple of minor variations you need to be aware of. I'll illustrate these in the next example.

TRY IT OUT: PRINTING ON A PRINTER

This program shows how you can get programs to output to a printer:

```
/* Program 10.12 Printing on a printer - where else? */
#include <stdio.h>
int main(void)
{
  fprintf(stdprn, "The barber shaves all those who do not"
                                        " shave themselves.");
  fprintf(stdprn, "\n\rQuestion: Who shaves the barber?\n\r");
  fprintf(stdprn, "\n\rAnswer: She doesn't need to shave.\f");
  return 0;
}
```

How It Works

The only oddities here are the new escape sequences `\r` and `\f`. The sequence `\n\r` is equivalent to newline/carriage return on a printer, and the `\f` is form-feed character, which produces a page eject on printers where this is necessary.

Summary

Although I chose the various specifications for formatting that you've seen in this chapter with the idea of them being as meaningful as possible, there are a lot of them. The only way you're going to become comfortable with them is through practice, and ideally this practice needs to take place in a real-world context. Understanding the various codes is one thing, but they'll probably become really familiar to you only once you've used them a few times in real programs. In the meantime, when you need a quick reminder you can always look them up in the summary Appendix D.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 10-1. Write a program that will read, store, and output the following five types of strings on separate lines, when one of each type of string is entered on a single line without spaces between the lines:

- *Type 1*: A sequence of lowercase letters followed by a digit (e.g., `number1`)
- *Type 2*: Two words that both begin with a capital letter and have a hyphen between them (e.g., `Seven-Up`)
- *Type 3*: A decimal value (e.g., `7.35`)
- *Type 4*: A sequence of upper- and lowercase letters and spaces (e.g., `Oliver Hardy`)
- *Type 5*: A sequence of any characters except spaces and digits (e.g., `floating-point`)

The following is a sample of input that should be read as five separate strings:

```
babylon5John-Boy3.14159Stan Laurel'Winner!'
```

Exercise 10-2. Write a program that will read the numerical values in the following line of input, and output the total:

```
  $3.50  ,  $4.75 , $9.95 , $2.50
```

Exercise 10-3. Define a function that will output an array of values of type `double` that is passed as an argument along with the number of elements in the array. The prototype of the function will be the following:

```
void show(double array[], int array_size, int field_width);
```

The values should be output 5 to a line, each with 2 decimal places after the decimal point and in a field width of 12. Use the function in a program that will output the values from 1.5 to 4.5 in steps of 0.3 (i.e., 1.5, 1.8, 2.1, and so on, up to 4.5).

Exercise 10-4. Define a function using the `getchar()` function that will read a string from `stdin` terminated by a character that is passed as the second argument to the function. Thus the prototype will be the following:

```
char *getString(char *buffer, char end_char);
```

The return value is the pointer that is passed as the first argument. Write a program to demonstrate the use of the function to read and output five strings that are from the keyboard, each terminated by a colon.

CHAPTER 11

Structuring Data

So far, you've learned how to declare and define variables that can hold various types of data, including integers, floating-point values, and characters. You also have the means to create arrays of any of these types and arrays of pointers to memory locations containing data of the types available to you. Although these have proved very useful, there are many applications in which you need even more flexibility.

For instance, suppose you want to write a program that processes data about breeding horses. You need information about each horse such as its name, its date of birth, its coloring, its height, its parentage, and so on. Some items are strings and some are numeric. Clearly, you could set up arrays for each data type and store them quite easily. However, this has limitations—for example, it doesn't allow you to refer to Dobbin's date of birth or Trigger's height particularly easily. You would need to synchronize your arrays by relating data items through a common index. Amazingly, C provides you with a better way of doing this, and that's what I'll discuss in this chapter.

In this chapter you'll learn the following:

- What structures are
- How to declare and define data structures
- How to use structures and pointers to structures
- How you can use pointers as structure members
- How to share memory between variables
- How to define your own data types
- How to write a program that produces bar charts from your data

Data Structures: Using struct

The keyword `struct` enables you to define a collection of variables of various types called a **structure** that you can treat as a single unit. This will be clearer if you see a simple example of a structure declaration:

```
struct horse
{
  int age;
  int height;
} Silver;
```

This example declares a **structure** type called `horse`. This isn't a variable name; it's a new type. This type name is usually referred to as a **structure tag**, or a **tag name**. The naming of the structure tag follows the same rules as for a variable name, which you should be familiar with by now.

■ **Note** It's legal to use the same name for a structure tag name and another variable. However, I don't recommend that you do this because it will make your code confusing and difficult to understand.

The variable names within the structure, `age` and `height`, are called **structure members**. In this case, they're both of type `int`. The members of the structure appear between the braces that follow the `struct` tag name `horse`.

In the example, an instance of the structure, called `Silver`, is declared at the same time that the structure is defined. `Silver` is a variable of type `horse`. Now, whenever you use the variable name `Silver`, it includes both members of the structure: the member `age` and the member `height`.

Let's look at the declaration of a slightly more complicated version of the structure type `horse`:

```
struct horse
{
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
} Dobbin = {
             24, 17, "Dobbin", "Trigger", "Flossie"
           };
```

Any type of variable can appear as a member of a structure, including arrays. As you can see, there are five members to this version of the structure type called `horse`: the integer members `age` and `height`, and the arrays `name`, `father`, and `mother`. Each member declaration is essentially the same as a normal variable declaration, with the type followed by the name and terminated by a semicolon. Note that initialization values can't be placed here because you aren't declaring variables; you're defining members of a type called `horse`. A structure type is a kind of specification or blueprint that can then be used to define variables of that type—in this example, type `horse`.

You define an instance of the structure `horse` after the closing brace of the structure definition. This is the variable `Dobbin`. Initial values are also assigned to the member variables of `Dobbin` in a manner similar to that used to initialize arrays, so initial values *can* be assigned when you define instances of the type `horse`.

In the declaration of the variable `Dobbin`, the values that appear between the final pair of braces apply, in sequence, to the member variable `age` (24), `height` (17), `name` (`"Dobbin"`), `father` (`"Trigger"`), and `mother` (`"Flossie"`). The statement is finally terminated with a semicolon. The variable `Dobbin` now refers to the complete collection of members included in the structure. The memory occupied by the structure `Dobbin` is shown in Figure 11-1, assuming variables of type `int` occupy 4 bytes. You can always find out the amount of memory that's occupied by a structure using the `sizeof` operator.

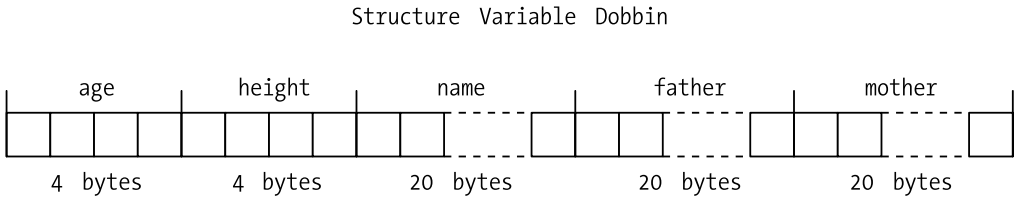


Figure 11-1. *Memory occupied by Dobbin*

Defining Structure Types and Structure Variables

You could have separated the declaration of the structure from the declaration of the structure variable. Instead of the statements you saw previously, you could have written the following:

```
struct horse
{
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
};

struct horse Dobbin = {
                        24, 17,"Dobbin", "Trigger", "Flossie"
                      };
```

You now have two separate statements. The first is the definition of the structure tag `horse`, and the second is a declaration of one variable of that type, `Dobbin`. Both the structure definition and the structure variable declaration statements end with a semicolon. The initial values for the members of the `Dobbin` structure tell you that the father of Dobbin is Trigger and his mother is Flossie.

You could also add a third statement to the previous two examples that would define another variable of type `horse`:

```
struct horse Trigger = {
                         30, 15, "Trigger", "Smith", "Wesson"
                       };
```

Now you have a variable `Trigger` that holds the details of the father of `Dobbin`, where it's clear that the ancestors of `Trigger` are `"Smith"` and `"Wesson"`.

Of course, you can also declare multiple structure variables in a single statement. This is almost as easy as declaring variables of one of the standard C types, for example

```
struct horse Piebald, Bandy;
```

declares two variables of type `horse`. The only additional item in the declaration, compared with standard types, is the keyword `struct`. You haven't initialized the values—to keep the statement simple—but in general it's wise to do so.

Accessing Structure Members

Now that you know how to define a structure and declare structure variables, you need to be able to refer to the members of a structure. A structure variable name is *not* a pointer. You need a special syntax to access the members.

You refer to a member of a structure by writing the structure variable name followed by a period, followed by the member variable name. For example, if you found that `Dobbin` had lied about his age and was actually much younger than the initializing value would suggest, you could amend the value by writing this:

```
Dobbin.age = 12;
```

The period between the structure variable name and the member name is actually an operator that is called the **member selection operator**. This statement sets the `age` member of the structure

Dobbin to 12. Structure members are just the same as variables of the same type. You can set their values and use them in expressions in the same way as ordinary variables.

TRY IT OUT: USING STRUCTURES

You can try out what you've learned so far about structures in a simple example that's designed to appeal to horse enthusiasts:

```
/* Program 11.1 Exercising the horse */
#include <stdio.h>

int main(void)
{
  /* Structure declaration */
  struct horse
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
  };

  struct horse My_first_horse;         /* Structure variable declaration   */

  /* Initialize the structure variable from input data */
  printf("Enter the name of the horse: " );
  scanf("%s", My_first_horse.name );            /* Read the horse's name   */

  printf("How old is %s? ", My_first_horse.name );
  scanf("%d", &My_first_horse.age );            /* Read the horse's age    */

  printf("How high is %s ( in hands )? ", My_first_horse.name );
  scanf("%d", &My_first_horse.height );         /* Read the horse's height */

  printf("Who is %s's father? ", My_first_horse.name );
  scanf("%s", My_first_horse.father );          /* Get the father's name   */

  printf("Who is %s's mother? ", My_first_horse.name );
  scanf("%s", My_first_horse.mother );          /* Get the mother's name   */

  /* Now tell them what we know */
  printf("\n%s is %d years old, %d hands high,",
       My_first_horse.name, My_first_horse.age, My_first_horse.height);
  printf(" and has %s and %s as parents.\n", My_first_horse.father,
                                          My_first_horse.mother );
  return 0;
}
```

Depending on what data you key in, you should get output approximating to the following:

```
Enter the name of the horse: Neddy
How old is Neddy? 12
How high is Neddy ( in hands )? 14
Who is Neddy's father? Bertie
Who is Neddy's mother? Nellie

Neddy is 12 years old, 14 hands high, and has Bertie and Nellie as parents.
```

How It Works

The way you reference members of a structure makes it very easy to follow what is going on in this example. You define the structure `horse` with this statement:

```
struct horse
{
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
};
```

The structure has two integer members, `age` and `height`, and three `char` array members, `name`, `father`, and `mother`. Because there's just a semicolon following the closing brace, no variables of type `horse` are declared here.

After defining the structure `horse`, you have the following statement:

```
struct horse My_first_horse;          /* Structure variable declaration   */
```

This declares one variable of type `horse`, which is `My_first_horse`. This variable has no initial values assigned in the declaration.

You then read in the data for the `name` member of the structure `My_first_horse` with this statement:

```
scanf("%s", My_first_horse.name );            /* Read the horse's name   */
```

No address of operator (&) is necessary here, because the `name` member of the structure is an array, so you implicitly transfer the address of the first array element to the function `scanf()`. You reference the member by writing the structure name, `My_first_horse`, followed by a period, followed by the name of the member, which is `name`. Other than the notation used to access it, using a structure member is the same as using any other variable.

The next value you read in is for the age of the horse:

```
scanf("%d", &My_first_horse.age );            /* Read the horse's age    */
```

This member is a variable of type `int`, so here you must use the & to pass the address of the structure member.

Note When you use the address of operator (&) for a member of a `struct` object, you place the & in front of the whole reference to the member, not in front of the member name.

The following statements read the data for each of the other members of the structure in exactly the same manner, prompting for the input in each case. Once input is complete, the values read are output to the display as a single line using the following statements:

```
printf("\n\n%s is %d years old, %d hands high,",
       My_first_horse.name, My_first_horse.age, My_first_horse.height);
  printf(" and has %s and %s as parents.", My_first_horse.father,
                                           My_first_horse.mother );
```

The long names that are necessary to refer to the members of the structure tend to make this statement appear complicated, but it's quite straightforward. You have the names of the member variables as the arguments to the function following the first argument, which is the standard sort of format control string that you've seen many times before.

Unnamed Structures

You don't have to give a structure a tag name. When you declare a structure and any instances of that structure in a single statement, you can omit the tag name. In the last example, instead of the structure declaration for type `horse`, followed by the instance declaration for `My_first_horse`, you could have written this statement:

```
struct
{                                          /* Structure declaration and... */
  int age;
  int height;
  char name[20];
  char father[20];
  char mother[20];
} My_first_horse;             /* ...structure variable declaration combined */
```

A serious disadvantage with this approach is that you can no longer define further instances of the structure in another statement. All the variables of this structure type that you want in your program must be defined in the one statement.

Arrays of Structures

The basic approach to keeping horse data is fine as far as it goes. But it will probably begin to be a bit cumbersome by the time you've accumulated 50 or 100 horses. You need a more stable method for handling a lot of horses. It's exactly the same problem that you had with variables, which you solved using an array. And you can do the same here: you can declare a `horse` array.

TRY IT OUT: USING ARRAYS OF STRUCTURES

Let's saddle up and extend the previous example to handle several horses:

```
/* Program 11.2   Exercising the horses */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  struct horse                        /* Structure declaration */
  {
    int age;
    int height;
```

```
 char name[20];
    char father[20];
    char mother[20];
  };

  struct horse My_horses[50];         /* Structure array declaration   */
  int hcount = 0;                     /* Count of the number of horses */
  char test = '\0';                   /* Test value for ending         */

  for(hcount = 0; hcount<50 ; hcount++ )
  {
    printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                          hcount?"nother " : "" );
    scanf(" %c", &test );
    if(tolower(test) == 'n')
       break;

    printf("\nEnter the name of the horse: " );
    scanf("%s", My_horses[hcount].name );  /* Read the horse's name */

    printf("\nHow old is %s? ", My_horses[hcount].name );
    scanf("%d", &My_horses[hcount].age );  /* Read the horse's age  */

    printf("\nHow high is %s ( in hands )? ", My_horses[hcount].name );
    /* Read the horse's height*/
    scanf("%d", &My_horses[hcount].height );

    printf("\nWho is %s's father? ", My_horses[hcount].name );
    /* Get the father's name */
     scanf("%s", My_horses[hcount].father );

    printf("\nWho is %s's mother? ", My_horses[hcount].name );

    /* Get the mother's name  */
    scanf("%s", My_horses[hcount].mother );
  }

  /* Now tell them what we know. */
  for(int i = 0 ; i<hcount ; i++ )
  {
    printf("\n\n%s is %d years old, %d hands high,",
             My_horses[i].name, My_horses[i].age, My_horses[i].height);
    printf(" and has %s and %s as parents.", My_horses[i].father,
                                             My_horses[i].mother );
  }
  return 0;
}
```

The output from this program is a little different from the previous example you saw that dealt with a single horse. The main addition is the prompt for input data for each horse. Once all the data for a few horses has been entered, or if you have the stamina, the data on 50 horses has been entered, the program outputs a summary of all the data that has been read in, one line per horse. The whole mechanism is stable and works very well in the mane (almost an unbridled success, you might say).

How It Works

In this version of equine data processing, you first declare the `horse` structure, and this is followed by the declaration

```
struct horse My_horses[50]; /* Structure array declaration   */
```

This declares the variable `My_horses`, which is an array of 50 `horse` structures. Apart from the keyword `struct`, it's just like any other array declaration.

You then have a `for` loop controlled by the variable `hcount`:

```
for(hcount = 0; hcount<50 ; hcount++ )
{
  ...
}
```

This creates the potential for the program to read in data for up to 50 horses. The loop control variable `hcount` is used to accumulate the total number of `horse` structures entered. The first action in the loop is in these statements:

```
printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                          hcount?"nother " : "" );
scanf(" %c", &test );
if(tolower(test) == 'n')
   break;
```

On each iteration the user is prompted to indicate if he or she wants to enter data for another horse by entering **Y** or **N**. The `printf()` statement for this uses the conditional operator to insert `"nother"` into the output on every iteration after the first. After reading the character that the user enters, using `scanf()`, the `if` statement executes a `break`, which immediately exits from the loop if the response is negative.

The succeeding sequence of `printf()` and `scanf()` statements are much the same as before, but there are two points to note in these. Look at this statement:

```
scanf("%s", My_horses[hcount].name );  /* Read the horse's name */
```

You can see that the method for referencing the member of one element of an array of structures is easier to write than to say! The structure array name has an index in square brackets, to which a period and the member name are appended. If you want to reference the third element of the `name` array for the fourth structure element, you would write `My_horses[3].name[2]`.

Note Of course, the index values start from 0, as with arrays of other types, so the fourth element of the structure array has the index value 3, and the third element of the member array is accessed by the index value 2.

Now look at this statement from the example:

```
scanf("%d", &My_horses[hcount].age );  /* Read the horse's age  */
```

Notice that the arguments to `scanf()` don't need the & for the string array variables, such as `My_horses[hcount].name`, but they *do* require them for the integer arguments `My_horses[hcount].age` and `My_horses[hcount].height`. It's very easy to forget the address of operator when reading values for variables like these.

Don't be misled at this point and think that these techniques are limited to equine applications. They can perfectly well be applied to porcine problems and also to asinine exercises.

Structures in Expressions

A structure member that is one of the built-in types can be used like any other variable in an expression. Using the structure from Program 11.2, you could write this rather meaningless computation:

```
My_horses[1].height = (My_horses[2].height + My_horses[3].height)/2;
```

I can think of no good reason why the height of one horse should be the average of two other horses' heights (unless there's some Frankenstein-like assembly going on) but it's a legal statement.

You can also use a complete structure element in an assignment statement:

```
My_horses[1] = My_horses[2];
```

This statement causes *all* the members of the structure `My_horses[2]` to be copied to the structure `My_horses[1]`, which means that the two structures become identical. The only other operation that's possible with a whole structure is to take its address using the `&` operator. You can't add, compare, or perform any other operations with a complete structure. To do those kinds of things, you must write your own functions.

Pointers to Structures

The ability to obtain the address of a structure raises the question of whether you can have pointers to a structure. Because you can take the address of a structure, the possibility of declaring a pointer to a structure does, indeed, naturally follow. You use the notation that you've already seen with other types of variables:

```
struct horse *phorse;
```

This declares a pointer, `phorse`, that can store the address of a structure of type `horse`.

You can now set `phorse` to have the value of the address of a particular structure, using exactly the same kind of statement that you've been using for other pointer types, for example

```
phorse = &My_horses[1];
```

Now `phorse` points to the structure `My_horses[1]`. You can immediately reference elements of this structure through your pointer. So if you want to display the name member of this structure, you could write this:

```
printf("\nThe name is %s.", (*phorse).name);
```

The parentheses around the dereferenced pointer are essential, because the precedence of the member selection operator (the period) is higher than that of the pointer-dereferencing operator *.

However, there's another way of doing this, and it's much more readable and intuitive. You could write the previous statement as follows:

```
printf("\nThe name is %s.", phorse->name );
```

So you don't need parentheses or an asterisk. You construct the operator -> from a minus sign immediately followed by the greater-than symbol. The operator is sometimes called the **pointer to member** operator for obvious reasons. This notation is almost invariably used in preference to the usual pointer-dereferencing notation you used at first, because it makes your programs so much easier to read.

Dynamic Memory Allocation for Structures

You have virtually all the tools you need to rewrite Program 11.2 with a much more economical use of memory. In the original version, you allocated the memory for an array of 50 `horse` structures, even when in practice you probably didn't need anything like that amount.

To create dynamically allocated memory for structures, the only tool that's missing is an array of pointers to structures, which is declared very easily, as you can see in this statement:

```
struct horse *phorse[50];
```

This statement declares an array of 50 pointers to structures of type `horse`. Only memory for the pointers has been allocated by this statement. You must still allocate the memory necessary to store the actual members of each structure that you need.

TRY IT OUT: USING POINTERS WITH STRUCTURES

You can see the dynamic allocation of memory for structures at work in the following example:

```
/* Program 11.3 Pointing out the horses */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>          /* For malloc() */

int main(void)
{
  struct horse              /* Structure declaration */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
  };

  struct horse *phorse[50]; /* pointer to structure array declaration */
  int hcount = 0;           /* Count of the number of horses          */
  char test = '\0';         /* Test value for ending input            */

  for(hcount = 0; hcount < 50 ; hcount++ )
  {
    printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                          hcount?"nother " : "" );
    scanf(" %c", &test );
    if(tolower(test) == 'n')
      break;

    /* allocate memory to hold a structure    */
    phorse[hcount] = (struct horse*) malloc(sizeof(struct horse));

    printf("\nEnter the name of the horse: " );
    scanf("%s", phorse[hcount]->name );  /* Read the horse's name */
```

```
    printf("\nHow old is %s? ", phorse[hcount]->name );
    scanf("%d", &phorse[hcount]->age );  /* Read the horse's age  */

    printf("\nHow high is %s ( in hands )? ", phorse[hcount]->name );
    scanf("%d", &phorse[hcount]->height ); /* Read the horse's height */

    printf("\nWho is %s's father? ", phorse[hcount]->name );
    scanf("%s", phorse[hcount]->father );  /* Get the father's name */

    printf("\nWho is %s's mother? ", phorse[hcount]->name );
    scanf("%s", phorse[hcount]->mother );  /* Get the mother's name */
  }

  /* Now tell them what we know. */
  for(int i = 0 ; i < hcount ; i++ )
  {
    printf("\n\n%s is %d years old, %d hands high,",
                phorse[i]->name, phorse[i]->age, phorse[i]->height);
    printf(" and has %s and %s as parents.",
                              phorse[i]->father, phorse[i]->mother);
    free(phorse[i]);
  }
  return 0;
}
```

The output should be exactly the same as that from Program 11.2, given the same input.

How It Works

This looks very similar to the previous version, but it operates rather differently. Initially, you don't have any memory allocated for any structures. The declaration

```
struct horse *phorse[50]; /* pointer to structure array declaration */
```

defines only 50 pointers to structures of type `horse`. You still have to find somewhere to put the structures to which these pointers are going to point:

```
phorse[hcount] = (struct horse*) malloc(sizeof(struct horse));
```

In this statement, you allocate the space for each structure as it's required. Let's have a quick reminder of how the `malloc()` function works. The `malloc()` function allocates the number of bytes specified by its argument and returns the address of the block of memory allocated as a pointer to type `void`. In this case, you use the `sizeof` operator to provide the value required.

It's very important to use `sizeof` when you need the number of bytes occupied by a structure. It doesn't necessarily correspond to the sum of the bytes occupied by each of its individual members, so you're likely to get it wrong if you try to work it out yourself.

Variables other than type `char` are often stored beginning at an address that's a multiple of 2 for 2-byte variables, a multiple of 4 for 4-byte variables, and so on. This is called **boundary alignment** and it has nothing to do with C but is a hardware requirement where it applies. Arranging variables to be stored in memory like this makes the transfer of data between the processor and memory faster. This arrangement can result in unused bytes occurring between member variables of different types, though, depending on their sequence. These have to be accounted for in the number of bytes allocated for a structure. Figure 11-2 presents an illustration of how this can occur.

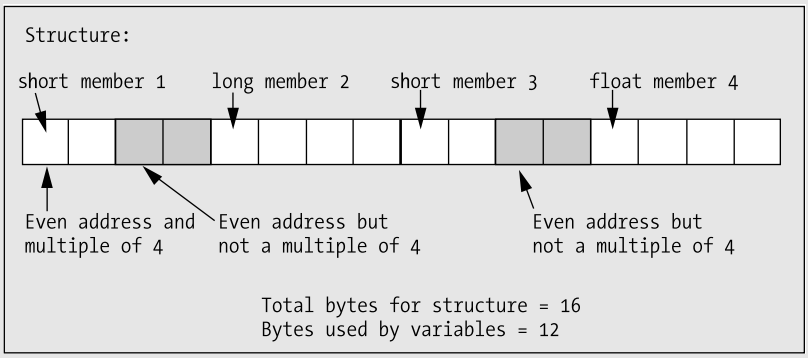


Figure 11-2. *The effect of boundary alignment on memory allocation*

As the value returned by `malloc()` is a pointer to `void`, you then cast this to the type you require with the expression `(struct horse*)`. This enables the pointer to be incremented or decremented correctly, if required.

```
scanf("%s", phorse[hcount]->name );  /* Read the horse's name     */
```

In this statement, you use the new notation for selecting members of a structure through a pointer. It's much clearer than `(*phorse[hcount]).name`. All subsequent references to members of a specific `horse` structure use this new notation.

Lastly in this program, you display a summary of all the data entered for each horse, freeing the memory as you go along.

More on Structure Members

So far, you've seen that any of the basic data types, including arrays, can be members of a structure. But there's more. You can also make a structure a member of a structure. Furthermore, not only can pointers be members of a structure, but also a pointer to a structure can be a member of a structure.

This opens up a whole new range of possibilities in programming with structures and, at the same time, increases the potential for confusion. Let's look at each of these possibilities in sequence and see what they have to offer. Maybe it won't be a can of worms after all.

Structures As Members of a Structure

At the start of this chapter, you examined the needs of horse breeders and, in particular, the necessity to manage a variety of details about each horse, including its name, height, date of birth, and so on. You then went on to look at Program 11.1, which carefully avoided date of birth and substituted age instead. This was partly because dates are messy things to deal with, as they're represented by three numbers and hold all the complications of leap years. However, you're now ready to tackle dates, using a structure that's a member of another structure.

You can define a structure type designed to hold dates. You can specify a suitable structure with the tag name `Date` with this statement:

```
struct Date
{
  int day;
  int month;
  int year;
};
```

Now you can define the structure `horse`, including a date-of-birth variable, like this:

```
struct horse
{
  struct Date dob;
  int height;
  char name[20];
  char father[20];
  char mother[20];
};
```

Now you have a single variable member within the structure that represents the date of birth of a horse, and this member is itself a structure. Next, you can define an instance of the structure `horse` with the usual statement:

```
struct horse Dobbin;
```

You can define the value of the member `height` with the same sort of statement that you've already seen:

```
Dobbin.height = 14;
```

If you want to set the date of birth in a series of assignment statements, you can use the logical extension of this notation:

```
Dobbin.dob.day = 5;
Dobbin.dob.month = 12;
Dobbin.dob.year = 1962;
```

You have a very old horse. The expression `Dobbin.dob.day` is referencing a variable of type `int`, so you can happily use it in arithmetic or comparative expressions. But if you use the expression `Dobbin.dob`, you would be referring to a `struct` variable of type `date`. Because this is clearly not a basic type but a structure, you can use it only in an assignment such as this:

```
Trigger.dob = Dobbin.dob;
```

This *could* mean that they're twins, but it doesn't guarantee it.

If you can find a good reason to do it, you can extend the notion of structures that are members of a structure to a structure that's a member of a structure that's a member of a structure. In fact, if you can make sense of it, you can continue with further levels of structure. Your C compiler is likely to provide for at least 15 levels of such convolution. But beware: if you reach this depth of structure nesting, you're likely to be in for a bout of repetitive strain injury just typing the references to members.

Declaring a Structure Within a Structure

You could declare the `Date` structure within the `horse` structure definition, as in the following code:

```
struct horse
{
  struct Date
  {
    int day;
    int month;
    int year;
  } dob;

  int height;
  char name[20];
  char father[20];
  char mother[20];
};
```

This has an interesting effect. Because the declaration is enclosed within the scope of the `horse` structure definition, it doesn't exist outside it, and so it becomes impossible to declare a `Date` variable external to the `horse` structure. Of course, each instance of a `horse` type variable would contain the `Date` type member, `dob`. But a statement such as this

```
struct date my_date;
```

would cause a compiler error. The message generated will say that the structure type `date` is undefined. If you need to use `date` outside the structure `horse`, its definition must be placed outside of the `horse` structure.

Pointers to Structures As Structure Members

Any pointer can be a member of a structure. This includes a pointer that points to a structure. A pointer structure member that points to the same type of structure is also permitted. For example, the `horse` type structure could contain a pointer to a `horse` type structure. Interesting, but is it of any use? Well, as it happens, yes.

TRY IT OUT: POINTERS TO STRUCTURES AS STRUCTURE MEMBERS

You can demonstrate a structure containing a pointer to a structure of the same type with a modification of the last example:

```
/* Program 11.4   Daisy chaining the horses */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main(void)
{
  struct horse                        /* Structure declaration       */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
```

```
  struct horse *next;              /* Pointer to next structure   */
};

struct horse *first = NULL;        /* Pointer to first horse      */
struct horse *current = NULL;      /* Pointer to current horse    */
struct horse *previous = NULL;     /* Pointer to previous horse   */

char test = '\0';                  /* Test value for ending input */

for( ; ; )
{
  printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                    first != NULL?"nother " : "" );
  scanf(" %c", &test );
  if(tolower(test) == 'n')
    break;

  /* Allocate memory for a structure */
  current = (struct horse*) malloc(sizeof(struct horse));

  if(first == NULL)
    first = current;               /* Set pointer to first horse  */

  if(previous != NULL)

    previous -> next = current; /* Set next pointer for previous horse */

  printf("\nEnter the name of the horse: ");
  scanf("%s", current -> name);    /* Read the horse's name       */

  printf("\nHow old is %s? ", current -> name);
  scanf("%d", &current -> age);    /* Read the horse's age        */

  printf("\nHow high is %s ( in hands )? ", current -> name );
  scanf("%d", &current -> height);  /* Read the horse's height    */

  printf("\nWho is %s's father? ", current -> name);
  scanf("%s", current -> father);   /* Get the father's name      */

  printf("\nWho is %s's mother? ", current -> name);
  scanf("%s", current -> mother);   /* Get the mother's name      */

  current->next = NULL;            /* In case it's the last...    */
  previous = current;              /* Save address of last horse  */
}

/* Now tell them what we know. */
current = first;                   /* Start at the beginning      */

while (current != NULL)      /* As long as we have a valid pointer */
{ /* Output the data*/
  printf("\n\n%s is %d years old, %d hands high,",
                    current->name, current->age, current->height);
```

```
 printf(" and has %s and %s as parents.", current->father,
                                          current->mother);
    previous = current;    /* Save the pointer so we can free memory */
    current = current->next;          /* Get the pointer to the next */
    free(previous);                   /* Free memory for the old one */
  }
  return 0;
}
```

This example should produce the same output as Program 11.3 (given the same input), but here you have yet another mode of operation.

How It Works

This time, not only do you have no space for structures allocated, but also you have only three pointers defined initially. These pointers are declared and initialized in these statements:

```
struct horse *first = NULL;          /* Pointer to first horse      */
struct horse *current = NULL;        /* Pointer to current horse    */
struct horse *previous = NULL;       /* Pointer to previous horse   */
```

Each of these pointers has been defined as a pointer to a `horse` structure. The pointer `first` is used solely to store the address of the first structure. The second and third pointers are working storage: `current` holds the address of the current `horse` structure that you're dealing with, and `previous` keeps track of the address of the previous structure that was processed.

You've added a member to the structure `horse` with the name `next`, which is a pointer to a `horse` type structure. This will be used to link together all the horses you have, where each `horse` structure will have a pointer containing the address of the next. The last structure will be an exception, of course: its `next` pointer will be `NULL`. The structure is otherwise exactly as you had previously. It's shown in Figure 11-3.

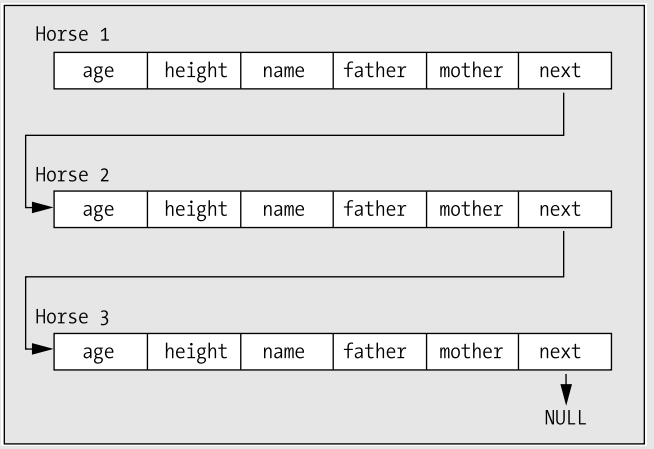


Figure 11-3. *A sequence of linked horses*

The input loop is the following:

```
for( ; ; )
{
  ...
}
```

The input loop is now an indefinite loop, because you don't have an array to worry about. You don't need to mess around with indexes. It's also unnecessary to keep count of how many sets of data are read in, so you don't need the variable `hcount` or the loop variable `i`. Because you allocate memory for each horse, you can just take them as they come.

The initial statements in the loop are the following:

```
printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                    first != NULL?"nother " : "" );
scanf(" %c", &test );
if(tolower(test) == 'n')
  break;
```

After the prompt, you exit from the loop if the response `'N'` or `'n'` is detected. Otherwise, you expect another set of structure members to be entered. You use the pointer `first` to get a slightly different prompt on the second and subsequent iterations, because the only time it will be `NULL` is on the first loop iteration.

Assuming you get past the initial question in the loop, you execute these statements:

```
current = (struct horse*) malloc(sizeof(struct horse));

if(first == NULL)
  first = current;                /* Set pointer to first horse  */

if(previous != NULL)
  previous -> next = current; /* Set next pointer for previous horse */
```

On each iteration, you allocate the memory necessary for the current structure. To keep things short, you don't check for a `NULL` return from `malloc()`, although you ought to do this in practice.

If the pointer `first` is `NULL`, you must be on the first loop iteration, and this must be the first structure about to be entered. Consequently, you set the pointer `first` to the pointer value that you've just obtained from `malloc()`, which is stored in the variable `current`. The address in `first` is the key to accessing the first horse in the chain. You can get to any of the others by starting with the address in `first` and then looking in the member pointer `next` to obtain the address of the next structure. You can step from there to the next structure and so on to any horse in the sequence.

The `next` pointer always needs to point to the next structure if there is one, but the address of the next structure can be determined only once you actually have the next structure. Therefore, on the second and subsequent iterations, you store the address of the current structure in the `next` member of the previous structure, whose address you'll have saved in `previous`. On the first iteration, the pointer `previous` will be `NULL` at this point, so of course you do nothing.

At the end of the loop, following all the input statements, you have these statements:

```
current->next = NULL;             /* In case it's the last...    */
previous = current;               /* Save address of last horse  */
```

The pointer `next` in the structure pointed to by `current`, which you're presently working with, is set to `NULL` in case this is the last structure and there's no next structure. If there is a next structure, this pointer `next` will be filled in on the next iteration. The pointer `previous` is set to `current` and is ready for the next iteration, when the `current` structure will indeed be the `previous` structure.

The strategy of the program is to generate a daisy chain of `horse` structures, in which the `next` member of each structure points to the next structure in the chain. The last is an exception because there's no next `horse`, so the `next` pointer contains `NULL`. This arrangement is called a **linked list**.

Once you have the `horse` data in a linked list, you process it by starting with the first structure and then getting the next structure through the pointer member `next`. When the pointer `next` is `NULL`, you know that you've reached the end of the list. This is how you generate the output list of all the input.

Linked lists are invaluable in applications in which you need to process an unknown number of structures, such as you have here. The main advantages of a linked list relate to memory usage and ease of handling. You occupy only the minimum memory necessary to store and process the list. Even though the memory used may be fragmented, you have no problem progressing from one structure to the next. As a consequence, in a practical situation in which you may need to deal with several different types of objects simultaneously, each can be handled using its own linked list, with the result that memory use is optimized. There is one small cloud associated with this—as there is with any silver lining—and it's that you pay a penalty in slower access to the data, particularly if you want to access it randomly.

The output process shows how a linked list is accessed as it steps through the linked list you've created with these statements:

```
current = first;                    /* Start at the beginning      */

while (current != NULL)      /* As long as we have a valid pointer */
{ /* Output the data*/
  printf("\n\n%s is %d years old, %d hands high,",
                    current->name, current->age, current->height);
  printf(" and has %s and %s as parents.", current->father,
                                          current->mother);
  previous = current;    /* Save the pointer so we can free memory */
  current = current->next;          /* Get the pointer to the next */
  free(previous);                   /* Free memory for the old one */
}
```

The `current` pointer controls the output loop, and it's set to `first` at the outset. Remember that the `first` pointer contains the address of the first structure in the list. The loop steps through the list, and as the members of each structure are displayed, the address stored in the member `next`, which points to the next structure, is assigned to `current`.

The memory for the structure displayed is then freed. It's obviously fairly essential that you only free the memory for a structure once you have no further need to reference it. It's easy to fall into the trap of putting the call of the function `free()` immediately after you've output all of the member values for the current structure. This would create some problems, because then you couldn't legally reference the current structure's `next` member to get the pointer to the next `horse` structure.

For the last structure in the linked list, the pointer `next` will contain `NULL` and the loop will terminate.

Doubly Linked Lists

A disadvantage of the linked list that you created in the previous example is that you can only go forward. However, a small modification of the idea gives you the **doubly linked list**, which will allow you to go through a list in either direction. The trick is to include an extra pointer in each structure to store the address of the previous structure in addition to the pointer to the next structure.

TRY IT OUT: DOUBLY LINKED LISTS

You can see a doubly linked list in action in a modified version of Program 11.4:

```
/* Program 11.5 Daisy chaining the horses both ways */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main(void)
{
  struct horse                        /* Structure declaration       */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
    struct horse *next;             /* Pointer to next structure     */
    struct horse *previous;         /* Pointer to previous structure */
  };

  struct horse *first = NULL;         /* Pointer to first horse      */
  struct horse *current = NULL;       /* Pointer to current horse    */
  struct horse *last = NULL;          /* Pointer to previous horse   */

  char test = '\0';                   /* Test value for ending input */

  for( ; ; )
  {
    printf("\nDo you want to enter details of a%s horse (Y or N)? ",
                                       first == NULL?"nother " : "");
    scanf(" %c", &test );
    if(tolower(test) == 'n')
      break;

    /* Allocate memory for each new horse structure */
    current = (struct horse*)malloc(sizeof(struct horse));

    if( first == NULL )
    {
      first = current;                /* Set pointer to first horse  */
     current->previous = NULL;
    }
    else
    {
      last->next = current;    /* Set next address for previous horse */
      current->previous = last; /* Previous address for current horse */
    }

    printf("\nEnter the name of the horse: ");
    scanf("%s", current -> name );    /* Read the horse's name   */
```

```
    printf("\nHow old is %s? ", current -> name);
    scanf("%d", &current -> age);     /* Read the horse's age    */

    printf("\nHow high is %s ( in hands )? ", current -> name);
    scanf("%d", &current -> height);  /* Read the horse's height */

    printf("\nWho is %s's father? ", current -> name);
    scanf("%s", current -> father);   /* Get the father's name   */

    printf("\nWho is %s's mother? ", current -> name);
    scanf("%s", current -> mother);   /* Get the mother's name   */

    current -> next = NULL;      /* In case it's the last horse..*/
    last = current;              /* Save address of last horse   */
  }

  /* Now tell them what we know. */
  while(current != NULL)         /* Output horse data in reverse order */
  {
    printf("\n\n%s is %d years old, %d hands high,",
              current->name, current->age, current->height);
    printf(" and has %s and %s as parents.", current->father,
                                             current->mother);
    last = current;       /* Save pointer to enable memory to be freed */
    current = current->previous; /* current points to previous in list */
    free(last);                 /* Free memory for the horse we output */
  }
  return 0;
}
```

For the same input, this program should produce the same output as before, except that the data on horses entered is displayed in reverse order to that of entry—just to show that you can do it.

How It Works

The initial pointer declarations are now as follows:

```
  struct horse *first = NULL;         /* Pointer to first horse      */
  struct horse *current = NULL;       /* Pointer to current horse    */
  struct horse *last = NULL;          /* Pointer to previous horse   */
```

You change the name of the pointer recording the `horse` structure entered on the previous iteration of the loop to `last`. This name change isn't strictly necessary, but it does help to avoid confusion with the structure member `previous`.

The structure `horse` is declared as follows:

```
  struct horse                        /* Structure declaration       */
  {
    int age;
    int height;
    char name[20];
    char father[20];
    char mother[20];
```

```
    struct horse *next;             /* Pointer to next structure     */
    struct horse *previous;         /* Pointer to previous structure */
  };
```

The `horse` structure now contains two pointers: one to point forward in the list, called `next`, the other to point backward to the preceding structure, called `previous`. This allows the list to be traversed in either direction, as you demonstrate by the fact that you output the data at the end of the program in reverse order.

Aside from the output, the only changes to the program are to add the statements that take care of the entries for the pointer structure member `previous`. In the beginning of the input loop you have the following:

```
    if( first == NULL )
    {
      first = current;               /* Set pointer to first horse   */
      current->previous = NULL;
    }
    else
    {
      last->next = current;    /* Set next address for previous horse */
      current->previous = last; /* Previous address for current horse */
    }
```

Here, you take the option of writing an `if` with an `else`, rather than the two `if`s you had in the previous version. The only material difference is setting the value of the structure member `previous`. For the first structure, `previous` is set to `NULL`, and for all subsequent structures it's set to the pointer `last`, whose value was saved on the preceding iteration.

The other change is at the end of the input loop:

```
last = current;                         /* Save address of last horse  */
```

This statement is added to allow the pointer `previous` in the next structure to be set to the appropriate value, which is the `current` structure that you're recording in the variable `last`.

The output process is virtually the same as in the previous example, except that you start from the last structure in the list and work back to the first.

Bit-Fields in a Structure

Bit-fields provide a mechanism that allows you to define variables that are each one or more binary bits within a single integer word, which you can nevertheless refer to explicitly with an individual member name for each one.

■ **Note** Bit-fields are used most frequently when memory is at a premium and you're in a situation in which you must use it as sparingly as possible. This is rarely the case these days so you won't see them very often. Bit-fields will slow your program down appreciably compared to using standard variable types. You must therefore assess each situation upon its merits to decide whether the memory savings offered by bit-fields are worth this price in execution speed for your programs. In most instances, bit-fields won't be necessary or even desirable, but you need to know about them.

An example of declaring a bit-field is shown here:

```
struct
{
  unsigned int flag1 : 1;
  unsigned int flag2 : 1;
  unsigned int flag3 : 2;
  unsigned int flag4 : 3;
} indicators;
```

This defines a variable with the name `indicators` that's an instance of an anonymous structure containing four bit-fields with the names `flag1` through `flag4`. These will all be stored in a single word, as illustrated in Figure 11-4.

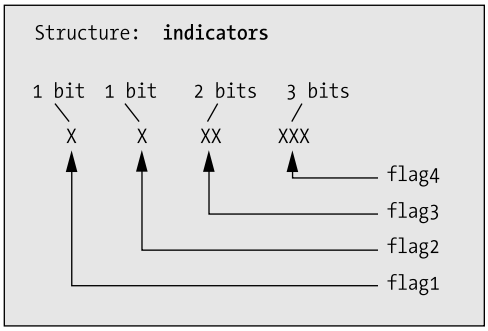


Figure 11-4. *Bit-fields in a structure*

The first two bit-fields, being a single bit specified by the 1 in their definition, can only assume the values 0 or 1. The third bit-field, `flag3`, has 2 bits and so it can have a value from 0 to 3. The last bit-field, `flag4`, can have values from 0 to 7, because it has 3 bits. These bit-fields are referenced in the same manner as other structure members, for example

```
indicators.flag4 = 5;
indicators.flag3 = indicators.flag1 = 1;
```

You'll rarely, if ever, have any need for this facility. I've included bit-fields here for the sake of completeness and for that strange off chance that one day bit-fields will be just what you need in a particularly tight memory situation.

Structures and Functions

Because structures represent such a powerful feature of the C language, their use with functions is very important. You'll now look at how you can pass structures as arguments to a function and how you can return a structure from a function.

Structures As Arguments to Functions

There's nothing unusual in the method for passing a structure as an argument to a function. It's exactly the same as passing any other variable. Analogous to the `horse` structure, you could create this structure:

```
struct family
{
  char name[20];
  int age;
  char father[20];
  char mother[20];
};
```

You could then construct a function to test whether two members of the type `family` are siblings:

```
bool siblings(struct family member1, struct family member2)
{
  if(strcmp(member1.mother, member2.mother) == 0)
    return true;
  else
    return false;
}
```

This function has two arguments, each of which is a structure. It simply compares the strings corresponding to the member `mother` for each structure. If they're the same, they are siblings and 1 is returned. Otherwise, they can't be siblings so 0 is returned. You're ignoring the effects of divorce, in vitro fertilization, cloning, and any other possibilities that may make this test inadequate.

Pointers to Structures As Function Arguments

Remember that a copy of the value of an argument is transferred to a function when it's called. If the argument is a large structure, it can take quite a bit of time, as well as occupying whatever additional memory the copy of the structure requires. Under these circumstances, you should use a pointer to a structure as an argument. This avoids the memory consumption and the copying time, because now only a copy of the pointer is made. The function will access the original structure directly through the pointer. More often than not, structures are passed to a function using a pointer, just for these reasons of efficiency. You could rewrite the `siblings()` function like this:

```
bool siblings(struct family *member1, struct family *member2)
{
  if(strcmp(member1->mother, member2->mother) == 0)
    return true;
  else
    return false;
}
```

Now, there is a downside to this. The pass-by-value mechanism provides good protection against accidental modification of values from within a called function. You lose this if you pass a pointer to a function. On the upside, if you don't need to modify the values pointed to by a pointer argument (you just want to access and use them, for instance), there's a technique for getting a degree of protection, even though you're passing pointers to a function.

Have another look at the last `siblings()` function. It doesn't need to modify the structures passed to it—in fact, it only needs to compare members. You could therefore rewrite it like this:

```
bool siblings(struct family const *pmember1, struct family const *pmember2)
{
  if(strcmp(pmember1->mother, pmember2->mother) == 0)
    return true;
  else
    return false;
}
```

You'll recall the const modifier from earlier in the book, where you used it to make a variable effectively a constant. This function declaration specifies the parameters as type "pointer to constant family structure." This implies that the structures pointed to by the pointers transferred to the function will be treated as constants within the function. Any attempt to change those structures will cause an error message during compilation. Of course, this doesn't affect their status as variables in the calling program, because the const keyword applies only to the values while the function is executing.

Note the difference between the previous definition of the function and this one:

```
bool siblings(struct family *const pmember1, struct family *const pmember2)
{
  if(strcmp(pmember1->mother, pmember2->mother) == 0)
    return true;
  else
    return false;
}
```

The indirection operator in each parameter definition is now in front of the keyword const, rather than in front of the pointer name as it was before. Does this make a difference? You bet it does. The parameters here are "constant pointers to structures of type family," not "pointers to constant structures." Now you're free to alter the structures themselves in the function, but you must not modify the addresses stored in the pointers. It's the pointers that are protected here, not the structures to which they point.

A Structure As a Function Return Value

There's nothing unusual about returning a structure from a function either. The function prototype merely has to indicate this return value in the normal way, for example:

```
struct horse my_fun(void);
```

This is a prototype for a function taking no arguments that returns a structure of type horse.

Although you can return a structure from a function like this, it's often more convenient to return a pointer to a structure. Let's explore this in more detail through a working example.

TRY IT OUT: RETURNING A POINTER TO A STRUCTURE

To demonstrate how returning a pointer to a structure works, you can rewrite the previous horse example in terms of humans and perform the input in a separate function:

```
/* Program 11.6 Basics of a family tree */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

struct Family *get_person(void);       /* Prototype for input function */

struct Date
{
  int day;
   int month;
  int year;
};
```

```
struct Family                          /* Family structure declaration */
{
  struct Date dob;
  char name[20];
  char father[20];
  char mother[20];
  struct Family *next;                /* Pointer to next structure    */
  struct Family *previous;           /* Pointer to previous structure */
};

int main(void)
{
  struct Family *first = NULL;        /* Pointer to first person      */
  struct Family *current = NULL;      /* Pointer to current person    */
  struct Family *last = NULL;         /* Pointer to previous person   */
  char more = '\0';                   /* Test value for ending input  */

  for( ; ; )
  {
    printf("\nDo you want to enter details of a%s person (Y or N)? ",
                                       first != NULL?"nother" : "");
    scanf(" %c", &more);
    if(tolower(more) == 'n')
      break;

    current = get_person();

    if(first == NULL)
    {
      first = current;                /* Set pointer to first Family  */
      last = current;                 /* Remember for next iteration  */
    }
    else
    {
      last->next = current;   /* Set next address for previous Family */
      current->previous = last;   /* Set previous address for current */
      last = current;             /* Remember for next iteration      */
    }
  }

  /* Now tell them what we know */

  /* Output Family data in reverse order */
  while (current  != NULL)
  {
    printf("\n%s was born %d/%d/%d, and has %s and %s as parents.",
              current->name, current->dob.day, current->dob.month,
              current->dob. year, current->father,  current->mother );

    last = current;      /* Save pointer to enable memory to be freed */
    current = current->previous;   /* current points to previous list */
    free(last);          /* Free memory for the Family we output      */
  }
```

```
 return 0;
}

/*   Function to input data on Family members   */
struct Family *get_person(void)
{
  struct Family *temp;          /* Define temporary structure pointer */

  /* Allocate memory for a structure */
  temp = (struct Family*) malloc(sizeof(struct Family));

  printf("\nEnter the name of the person: ");
  scanf("%s", temp -> name );         /* Read the Family's name       */

  printf("\nEnter %s's date of birth (day month year); ", temp->name);
  scanf("%d %d %d", &temp->dob.day, &temp->dob.month, &temp->dob.year);

  printf("\nWho is %s's father? ", temp->name );
  scanf("%s", temp->father );         /* Get the father's name        */

  printf("\nWho is %s's mother? ", temp->name );
  scanf("%s", temp -> mother );       /* Get the mother's name        */

  temp->next = temp->previous = NULL; /* Set pointers to NULL         */

  return temp;                  /* Return address of Family structure */
}
```

How It Works

Although this looks like a lot of code, you should find this example quite straightforward. It operates similarly to the previous example, but it's organized as two functions instead of one.

The first structure declaration is the following:

```
struct Date
{
  int day;
  int month;
  int year;
};
```

This defines a structure type `Date` with three members, `day`, `month`, and `year`, which are all declared as integers. No instances of the structure are declared at this point. The definition precedes all the functions in the source file so it is accessible from within any function that appears subsequently in the file.

The next structure declaration is the following:

```
struct Family                         /* Family structure declaration  */
{
  struct Date dob;
  char name[20];
  char father[20];
  char mother[20];
  struct Family *next;                /* Pointer to next structure    */
  struct Family *previous;           /* Pointer to previous structure */
};
```

This defines a structure type `Family`, which has a `Date` type structure as its first member. It then has three conventional `char` arrays as members. The last two members are pointers to structures. They're intended to allow a doubly linked list to be constructed, being pointers to the next and previous structures in the list, respectively.

The fact that both structure declarations are external to all the functions and are therefore available globally is an important difference between this and the previous examples. This is necessary here because you want to define `Family` structure variables in both the functions `main()` and `get_person()`.

■ **Note** Only the specification of the structure type is accessible globally. All the variables of type `Family` declared within each function are local in scope to the function in which they're declared.

The function `get_person()` has this prototype:

```
struct Family *get_person(void);  /* Prototype for input function   */
```

This indicates that the function accepts no arguments but returns a pointer to a `Family` structure.

The process parallels the operation of Program 11.5, with the differences that you have global structure type declarations and you input a structure within a separate function.

After verifying that the user wants to enter data by checking his or her response in `more`, the function `main()` calls the function `get_person()`. Within the function `get_person()`, you declare this pointer:

```
struct Family *temp;          /* Define temporary structure pointer */
```

This is a "pointer to a `Family` type structure" and it has local scope. The fact that the declaration of the structure type is global has no bearing on the scope of actual instances of the structure. The scope of each instance that you declare will depend on where the declaration is placed in the program.

The first action within the function `get_person()` is the following:

```
temp = (struct Family*) malloc(sizeof(struct Family));
```

This call to `malloc()` obtains sufficient memory to store a structure of type `Family` and stores the address that's returned in the pointer variable, `temp`. Although `temp` is local and will go out of scope at the end of the function `get_person()`, the memory allocated by `malloc()` is more permanent. It remains until you free it yourself within the program somewhere, or until you exit from the program completely.

The function `get_person()` reads in all the basic data for a person and stores that data in the structure pointed to by `temp`. As it stands, the function will accept any values for the date, but in a real situation you would include code for data validity checking. You could verify that the month value is from 1 to 12 and the day value is valid for the month entered. Because a birth date is being entered, you might verify that it isn't in the future.

The last statement in the function `get_person()` is the following:

```
return temp;                  /* Return address of Family structure */
```

This returns a copy of the pointer to the structure that it has created. Even though `temp` will no longer exist after the return, the address that it contains that points to the memory block obtained from `malloc()` is still valid.

Back in `main()`, the pointer that's returned is stored in the variable `current` and is also saved in the variable `first` if this is the first iteration. You do this because you don't want to lose track of the first structure in the list. You also save the pointer `current` in the variable `last`, so that on the next iteration you can fill in the backward pointer member, `previous`, for the current person whose data you've just obtained.

After all the input data has been read, the program outputs a summary to the screen in reverse order, in a similar fashion to the previous examples.

An Exercise in Program Modification

Perhaps we ought to produce an example combining both the use of pointers to structures as arguments and the use of pointers to structures as return values. You can declare some additional pointers, p_to_pa and p_to_ma, in the structure type Family in the previous example, Program 11.6, by changing that structure declaration as follows:

```
struct Family                          /* Family structure declaration  */
{
  struct Date dob;
  char name[20];
  char father[20];
  char mother[20];
  struct Family *next;                 /* Pointer to next structure     */
  struct Family *previous;             /* Pointer to previous structure */
  struct Family *p_to_pa;              /* Pointer to father structure   */
  struct Family *p_to_ma;              /* Pointer to mother structure   */
};
```

Now you can record the addresses of related structures in the pointer members p_to_pa and p_to_ma. You'll need to set them to NULL in the get_person() function by adding the following statement just before the return statement:

```
  temp->p_to_pa = temp->p_to_ma = NULL;       /* Set pointers to NULL  */
```

You can now augment the program with some additional functions that will fill in your new pointers p_to_pa and p_to_ma once data for everybody has been entered. You could code this by adding two functions. The first function, set_ancestry(), will accept pointers to Family structures as arguments and check whether the structure pointed to by the second argument is the father or mother of the structure pointed to by the first argument. If it is, the appropriate pointer will be updated to reflect this, and true will be returned; otherwise false will be returned. Here's the code:

```
bool set_ancestry(struct Family *pmember1, struct Family *pmember2)
{
  if(strcmp(pmember1->father, pmember2->name) == 0)
  {
    pmember1->p_to_pa = pmember2;
    return true;
  }

  if( strcmp(pmember1->mother, pmember2->name) == 0)
  {
    pmember1->p_to_ma = pmember2;
    return true;
  }
  else
    return false;
}
```

The second function will test all possible relationships between two Family structures:

```
/* Fill in pointers for mother or father relationships */
bool related (struct Family *pmember1, struct Family *pmember2)
{
  return set_ancestry(pmember1, pmember2) ||
                           set_ancestry(pmember2, pmember1);
}
```

The related() function calls set_ancestry() twice in the return statement to test all possibilities of relationship. The return value will be true if either of the calls to set_ancestry() return the value true. A calling program can use the return value from related() to determine whether a pointer has been filled in.

■ **Note** Because you use the library function strcmp() here, you must add an #include directive for <string.h> at the beginning of the program. You'll also need an #include directive for the <stdbool.h> header because you use the bool type and the values true and false.

You now need to add some code to the main() function that you created in Program 11.6 to use the function related() to fill in all the pointers in all the structures where valid addresses can be found. You can insert the following code into main() directly after the loop that inputs all the initial data:

```
current = first;

while(current->next != NULL) /* Check for relation for each person in */
{                            /* the list up to second to last         */
  int parents = 0;        /* Declare parent count local to this block */
  last = current->next;   /* Get the pointer to the next              */

  while(last != NULL)     /* This loop tests current person           */
  {                       /* against all the remainder in the list    */
    if(related(current, last))          /* Found a parent ?           */
      if(++parents == 2)     /* Yes, update count and check it        */
        break;               /* Exit inner loop if both parents found */

    last = last->next;       /* Get the address of the next           */
  }
  current = current->next;   /* Next in the list to check             */
}

/* Now tell them what we know etc. */
/* rest of output code etc.  ...   */
```

This is a relatively self-contained block of code to fill in the parent pointers where possible. Starting with the first structure, a check is made with each of the succeeding structures to see if a parent relationship exists. The checking stops for a given structure if two parents have been found (which would have filled in both pointers) or the end of the list is reached.

Of necessity, some structures will have pointers where the values can't be updated. Because you don't have an infinite list, and barring some very strange family history, there will always be someone whose parent records aren't included. The process will take each of the structures in the list in turn and check it against all the following structures to see if they're related, at least up to the point where the two parents have been discovered.

Of course, you also need to insert prototypes for the functions related() and set_ancestry() at the beginning of the program, immediately after the prototype for the function get_person(). These prototypes would look like this:

```
bool related(struct Family *pmember1, struct Family *pmember2);
bool set_ancestry(struct Family *pmember1, struct Family *pmember2);
```

To show that the pointers have been successfully inserted, you can extend the final output to display information about the parents of each person by adding some additional statements immediately after the last `printf()`. You can also amend the output loop to start from `first` so the output loop will thus be as follows:

```
/* Output Family data in correct order */
current = first;

while (current != NULL)             /* Output Family data in correct order */
{
  printf("\n%s was born %d/%d/%d, and has %s and %s as parents.",
                current->name, current->dob.day, current->dob.month,
             current->dob. year, current->father,  current->mother);
  if(current->p_to_pa != NULL )
    printf("\n\t%s's birth date is %d/%d/%d  ",
               current->father, current->p_to_pa->dob.day,
                                current->p_to_pa->dob.month,
                                current->p_to_pa->dob.year);
  if(current->p_to_ma != NULL)
    printf("and %s's birth date is %d/%d/%d.\n  ",
               current->mother, current->p_to_ma->dob.day,
                                current->p_to_ma->dob.month,
                                current->p_to_ma->dob.year);

  current = current->next;          /* current points to next in list      */
}
```

This should then produce the dates of birth of both parents for each person using the pointers to the parents' structures, but only if the pointers have been set to valid addresses. Note that you don't free the memory in the loop. If you do this, the additional statements to output the parents' dates of birth will produce junk output when the parent structure appears earlier in the list. So finally, you must add a separate loop at the end of `main()` to delete the memory when the output is complete:

```
/* Now free the memory */
current = first;
while(current != NULL)
{
  last = current;     /* Save pointer to enable memory to be freed */
  current = current->next; /* current points to next in list       */
  free(last);         /* Free memory for last                      */
}
```

If you've assembled all the pieces into a new example, you should have a sizeable new program to play with. Here's the sort of output that you should get:

```
Do you want to enter details of a person (Y or N)? y

Enter the name of the person: Jack

Enter Jack's date of birth (day month year); 1 1 65

Who is Jack's father? Bill

Who is Jack's mother? Nell

Do you want to enter details of another person (Y or N)? y

Enter the name of the person: Mary

Enter Mary's date of birth (day month year); 3 3 67

Who is Mary's father? Bert

Who is Mary's mother? Moll

Do you want to enter details of another person (Y or N)? y

Enter the name of the person: Ben

Enter Ben's date of birth (day month year); 2 2 89

Who is Ben's father? Jack

Who is Ben's mother? Mary

Do you want to enter details of another  person (Y or N)? n

Jack was born 1/1/65, and has Bill and Nell as parents.
Mary was born 3/3/67, and has Bert and Moll as parents.
Ben was born 2/2/89, and has Jack and Mary as parents.
        Jack's birth date is 1/1/65  and Mary's birth date is 3/3/67.
```

You could try to modify the program to output everybody in chronological order or possibly work out how many offspring each person has.

Binary Trees

A binary tree is a very useful way of organizing data because you can arrange that the data that you have stored in the tree can be extracted from the tree in an ordered fashion. A binary tree is also a very interesting mechanism because it is basically very simple. Implementing a binary tree can involve recursion as well as dynamic memory allocation. We'll also use pointers to pass structures around.

A binary tree consists of a set of interconnected elements called **nodes**. The starting node is the base of the tree and is called a **root node**, as shown in Figure 11-5.

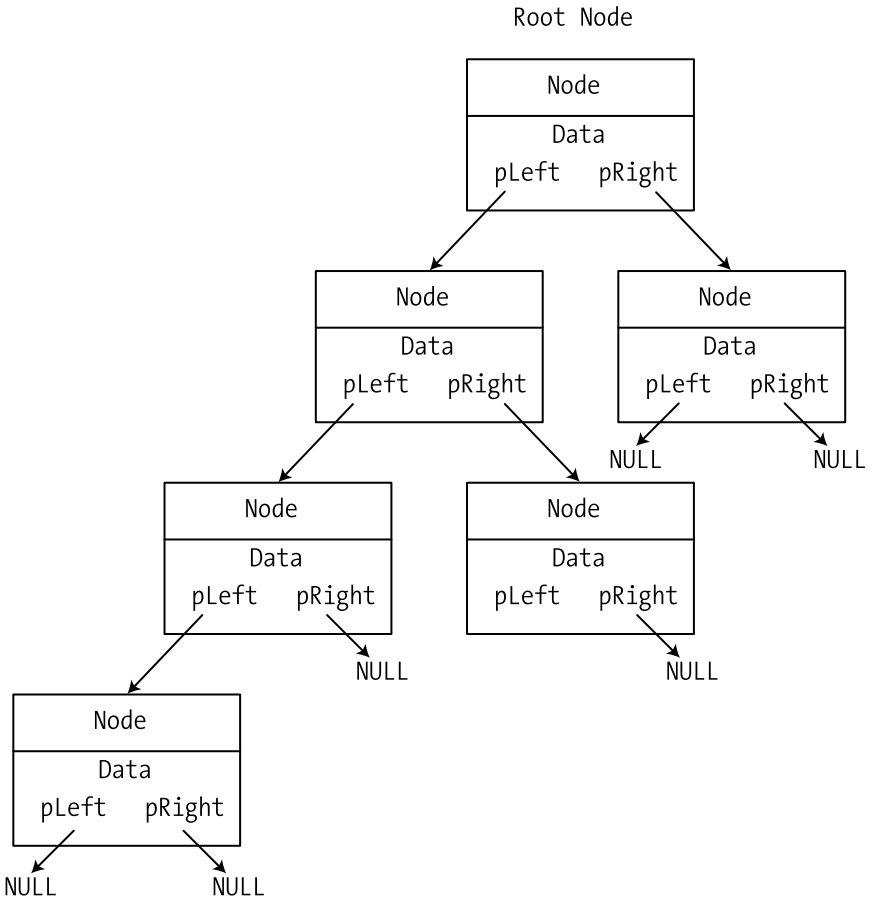


Figure 11-5. *The structure of a binary tree*

Each node typically contains an item of data, plus pointers to two subsidiary nodes, a **left node** and a **right node**. If either subsidiary node does not exist, the corresponding pointer is `NULL`. A node may also include a counter that records when there are duplicates of data items within the tree.

A `struct` makes it very easy to represent a binary tree node. Here's an example of a `struct` that defines nodes that store integers of type `long`:

```
struct Node
{
  long item;                /* The data item           */
  int count;                /* Number of copies of item */
  struct Node *pLeft;       /* Pointer to left node     */
  struct Node *pRight;      /* Pointer to right node    */
};
```

When a data item is to be added to the binary tree and that item already exists somewhere, a new node is not created; the `count` member of the existing node is simply incremented by 1. I'll use the previous definition of a `struct` that represents a node holding an integer when we get to creating a binary tree in practice in Program 11.7, which is the next working example in this chapter.

Ordering Data in a Binary Tree

The way in which you construct the binary tree will determine the order of data items within the tree. Adding a data item to a tree involves comparing the item to be added with the existing items in the tree. Typically items are added so that for every node, the data item stored in the subsidiary left node when it exists is less than the data item in the current node, and the data item stored in the right node when it exists is greater than the item stored in the node. An example of a tree containing an arbitrary sequence of integers is shown in Figure 11-6.

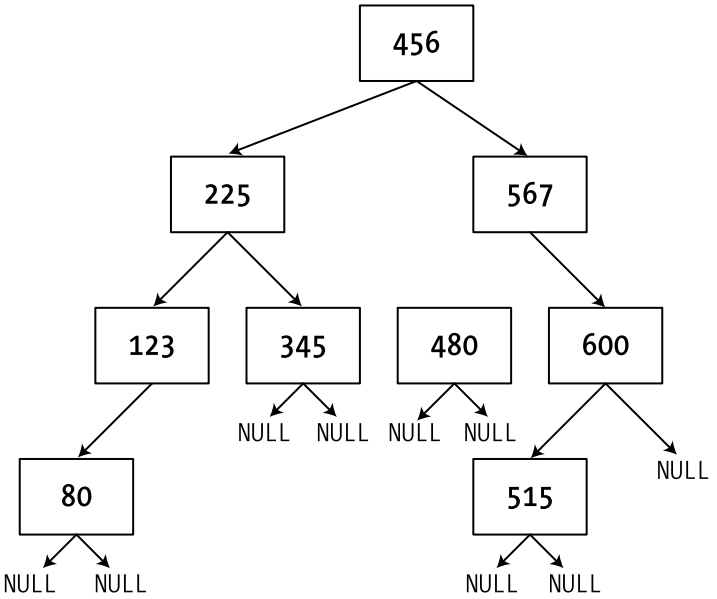


Figure 11-6. *A binary tree storing integers*

The structure of the tree will depend on the sequence in which the data items are added to the tree. Adding a new item involves comparing the data item with the values of the nodes in the tree starting with the root node. If it's less than a given node you then inspect the left subsidiary node, whereas if it's greater you look at the right subsidiary node. This process continues until either you find a node with an equal value, in which case you just update the count for that node, or you arrive at a left or right node pointer that is `NULL`, and that's where the new node is placed.

Constructing a Binary Tree

The starting point is the creation of the root node. All nodes will be created in the same way so the first step is to define a function that creates a node from a data item. I will assume we are creating a tree to store integers, so the `struct` definition you saw earlier will apply. Here's the definition of a function that will create a node:

```
struct Node *createnode(long value)
{
  /* Allocate memory for a new node */
  struct Node *pNode = (struct Node *)malloc(sizeof(struct Node));
```

```
  pNode->item = value;                      /* Set the value          */
  pNode->count = 1;                         /* Set the count          */
  pNode->pLeft = pNode->pRight = NULL;      /* No left or right nodes */
  return pNode;
}
```

The function allocates memory for the new `Node` structure and sets the `item` member to `value`. The `count` member is the number of duplicates of the value in the node so in the first instance this has to be 1. There are no subsidiary nodes at this point so the `pLeft` and `pRight` members are set to `NULL`. The function returns a pointer to the `Node` object that has been created.

To create the root node for a new binary tree you can just use this function, like this:

```
long newvalue;
printf("Enter the node value: ");
scanf(" %ld", newvalue);
struct Node *pRoot = createnode(newvalue);
```

After reading the value to be stored from the keyboard, you call the `createnode()` function to create a new node on the heap. Of course, you must not forget to release the memory for the nodes when you are done.

Working with binary trees is one of the areas where recursion really pays off. The process of inserting a node involves inspecting a succession of nodes in the same way, which is a strong indicator that recursion may be helpful. You can add a node to a tree that already exists with the following function:

```
/* Add a new node to the tree */
struct Node *addnode(long value, struct Node* pNode)
{
  if(pNode == NULL)                         /* If there's no node             */
    return createnode(value);               /* ...create one and return it    */

  if(value ==pNode->item)
  {                                         /* Value equals current node      */
    ++pNode->count;                         /* ...so increment count and      */
    return pNode;                           /* ...return the same node        */
  }

  if(value < pNode->item)                   /* If less than current node value */
  {
    if(pNode->pLeft == NULL)                /* and there's no left node       */
    {
      pNode->pLeft = createnode(value);     /* create a new left node and     */
      return pNode->pLeft;                  /* return it.                     */
    }
    else                                    /* If there is a left node...     */
      return addnode(value, pNode->pLeft); /* add value via the left node    */
  }
  else                                      /* value is greater than current  */
  {
    if(pNode->pRight == NULL)               /* so the same process with       */
    {                                       /* the right node.                */
      pNode-> pRight = createnode(value);
      return pNode-> pRight;
    }
```

```
    else
      return addnode(value, pNode-> pRight);
  }
}
```

The arguments to the `addnode()` function when you call it in the first instance are the value to be stored in the tree and the address of the root node. If you pass `NULL` as the second argument it will create and return a new node so you could also use this function to create the root node. When there is a root node passed as the second argument, there are three situations to deal with:

1. If `value` equals the value in the current node, no new node needs to be created, so you just increment the count in the current node and return it.
2. If `value` is less than the value in the current node, you need to inspect the left subsidiary node. If the left node pointer is `NULL`, you create a new node to hold `value` and make it the left subsidiary node. If the left node exists, you call `addnode()` recursively with the pointer to the left subsidiary node as the second argument.
3. If value is greater than the value in the current node, you proceed with the right node in the same way as with the left node.

Whatever happens within the recursive function calls, the function will return a pointer to the node where value is inserted. This may be a new node or one of the existing nodes if value is already in the tree somewhere.

You could construct a complete binary tree storing an arbitrary number of integers with the following code:

```
long newvalue = 0;
struct Node *pRoot = NULL;
char answer = 'n';
do
{
  printf("Enter the node value: ");
  scanf(" %ld", &newvalue);
  if(pRoot == NULL)
    pRoot = createnode(newvalue);
  else
  addnode(newvalue, pRoot);

  printf("\nDo you want to enter another (y or n)? ");
  scanf(" %c", &answer);
} while(tolower(answer) == 'y');
```

The `do-while` loop constructs the complete tree including the root node. On the first iteration `pRoot` will be `NULL`, so the root node will be created. All subsequent iterations will add nodes to the existing tree.

Traversing a Binary Tree

You can traverse a binary tree to extract the contents in ascending or descending sequence. I'll discuss how you extract the data in ascending sequence and you'll see how you can produce an descending sequence by analogy. At first sight it seems a complicated problem to extract the data from the tree because of its arbitrary structure but using recursion makes it very easy.

I'll start the explanation of the process by stating the obvious: the value of the left subnode is always less than the current node, and the value of the current node is always less than that of the

right subnode. You can conclude from this that the basic process is to extract the values in the sequence: left subnode, followed by current node, followed by right subnode. Of course, where the subnodes have subnodes, the process has to be applied to those too, from left through current to right. It will be easy to see how this works if we look at some code.

Suppose we want to simply list the integer values contained in our binary tree in ascending sequence. The function to do that looks like this:

```
/* List the node values in ascending sequence */
void listnodes(struct Node *pNode)
{
  if(pNode->pLeft != NULL)
    listnodes(pNode->pLeft);           /* List nodes in the left subtree */

  for(int i = 0; i<pNode->count ; i++)
    printf("\n%10ld", pNode->item);    /* Output the current node value  */

  if(pNode->pRight != NULL)
    listnodes(pNode->pRight);          /* List nodes in the right subtree */
}
```

It consists of three simple steps:

1. If the left subnode exists, you call `listnodes()` recursively for that node.
2. Write the value for the current node.
3. If the right subnode exists, call `listnodes()` recursively for that node.

The first step repeats the process for the left subnode of the root if it exists, so the whole subtree to the left will be written before the value of the current node. The value of the current node is repeated count times in the output to reflect the number of duplicate values for that node. The values for the entire right subtree to the root node will be written after the value for the current node. You should be able to see that this happens for every node in the tree, so the values will be written in ascending sequence. All you have to do to output the values in ascending sequence is to call `listnodes()` with the root node pointer as the argument, like this:

```
listnodes(pRoot);          /* Output the contents of the tree */
```

In case you find it hard to believe that such a simple function will output all the values in any binary tree of integers you care to construct, let's see it working.

TRY IT OUT: SORTING USING A BINARY TREE

This example drags together the code fragments you have already seen:

```
/* Program 11.7 Sorting integers using a binary tree */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

/* Function prototypes */
struct Node *createnode(long value);                  /* Create a tree node */
struct Node *addnode(long value, struct Node* pNode); /* Insert a new node  */
void listnodes(struct Node *pNode);                   /* List all nodes     */
void freenodes(struct Node *pNode);                   /* Release memory     */
```

```
/* Defines a node in a binary tree sotring integers */
struct Node
{
  long item;                /* The data item            */
  int count;                /* Number of copies of item */
  struct Node *pLeft;       /* Pointer to left node     */
  struct Node *pRight;      /* Pointer to right node    */
};

/* Function main - execution starts here */
int main(void)
{
  long newvalue = 0;
  struct Node *pRoot = NULL;
  char answer = 'n';
  do
  {
    printf("Enter the node value: ");
    scanf(" %ld", &newvalue);
    if(pRoot == NULL)
      pRoot = createnode(newvalue);
    else
    addnode(newvalue, pRoot);
    printf("\nDo you want to enter another (y or n)? ");
    scanf(" %c", &answer);
  } while(tolower(answer) == 'y');

  printf("The values in ascending sequence are: ");
  listnodes(pRoot);          /* Output the contents of the tree */
  freenodes(pRoot);          /* Release the heap memory         */

  return 0;
}

struct Node *createnode(long value)
{
  struct Node *pNode = (struct Node *)malloc(sizeof(struct Node));
  pNode->item = value;                      /* Set the value          */
  pNode->count = 1;                         /* Set the count          */
  pNode->pLeft = pNode->pRight = NULL;      /* No left or right nodes */
  return pNode;
}

/* Add a new node to the tree */
struct Node *addnode(long value, struct Node* pNode)
{
  if(pNode == NULL)                         /* If there's no node             */
    return createnode(value);               /* ...create one and return it    */

  if(value ==pNode->item)
  {                                         /* Value equals current node      */
    ++pNode->count;                         /* ...so increment count and      */
    return pNode;                           /* ...return the same node        */
  }
```

```
  if(value < pNode->item)                 /* If less than current node value */
  {
    if(pNode->pLeft == NULL)              /* and there's no left node        */
    {
      pNode->pLeft = createnode(value);   /* create a new left node and      */
      return pNode->pLeft;                /* return it.                      */
    }
    else                                  /* If there is a left node...      */
      return addnode(value, pNode->pLeft); /* add value via the left node     */
  }
  else                                    /* value is greater than current   */
  {
    if(pNode->pRight == NULL)             /* so the same process with        */
    {                                     /* the right node.                 */
      pNode-> pRight = createnode(value);
      return pNode-> pRight;
    }
    else
      return addnode(value, pNode-> pRight);
  }
}

/* List the node values in ascending sequence */
void listnodes(struct Node *pNode)
{
  if(pNode->pLeft != NULL)
    listnodes(pNode->pLeft);

  for(int i = 0; i<pNode->count ; i++)
    printf("\n%10ld", pNode->item);

  if(pNode->pRight != NULL)
    listnodes(pNode->pRight);
}

/* Release memory allocated to nodes */
void freenodes(struct Node * pNode)
{
  if(pNode == NULL)                   /* If there's no node...        */
    return;                           /* we are done.                 */

  if(pNode->pLeft != NULL)            /* If there's a left sub-tree   */
    freenodes(pNode->pLeft);          /* free memory for those nodes. */

  if(pNode->pRight != NULL)           /* If there's a right sub-tree  */
    freenodes(pNode->pRight);         /* free memory for those nodes. */

  free(pNode);                        /* Free current node memory     */
}
```

Here is some typical output from this example:

```
Enter the node value: 56

Do you want to enter another (y or n)? y
Enter the node value: 33

Do you want to enter another (y or n)? y
Enter the node value: 77

Do you want to enter another (y or n)? y
Enter the node value: -10

Do you want to enter another (y or n)? y
Enter the node value: 100

Do you want to enter another (y or n)? y
Enter the node value: -5

Do you want to enter another (y or n)? y
Enter the node value: 200

Do you want to enter another (y or n)? n
The values in ascending sequence are:
       -10
        -5
        33
        56
        77
       100
       200
```

How It Works

The `do-while` loop in `main()` constructs the binary tree from the values that are entered in the way I discussed earlier. The loop continues as long as you enter `'y'` or `'Y'` when prompted. Calling `listnodes()` with the address of the root node as the argument outputs all the values in the tree in ascending sequence. You then call the `freenodes()` function to release the memory that is allocated for the nodes for the tree.

The `freenodes()` function is the only new code in the example. This is another recursive function that works in a similar way to the `listnodes()` function. It is essential to delete the memory for the subsidiary nodes of each node before freeing the memory for the node itself, because once you have freed a memory block, it could be used immediately by some other program that is executing concurrently. This means that the addresses for the subsidiary nodes are effectively unavailable once the memory has been released. The function therefore always calls `freenodes()` for the subsidiary node pointers if they are not `NULL` before releasing the memory for the current node.

You can construct binary trees to store any kind of data including `struct` objects and strings. If you want to organize strings in a binary tree for example, you could use a pointer in each node to refer to the string rather than making copies of the strings within the tree.

Sharing Memory

You've already seen how you can save memory through the use of bit-fields, which are typically applied to logical variables. C has a further capability that allows you to place several variables in the

same memory area. This can be applied somewhat more widely than bit-fields when memory is short, because circumstances frequently arise in practice in which you're working with several variables, but only one of them holds a valid value at any given moment.

Another situation in which you can share memory between a number of variables to some advantage is when your program processes a number of different kinds of data record, but only one kind at a time, and the kind to be processed is determined at execution time. A third possibility is that you want to access the same data at different times, and assume it's of a different type on different occasions. You might have a group of variables of numeric data types, for instance, that you want to treat as simply an array of type `char` so that you can move them about as a single chunk of data.

Unions

The facility in C that allows the same memory area to be shared by a number of different variables is called a **union**. The syntax for declaring a union is similar to that used for structures, and a union is usually given a tag name in the same way. You use the keyword `union` to define a union. For example, the following statement declares a union to be shared by three variables:

```
union u_example
{
  float decval;
  int *pnum;
  double my_value;
} U1;
```

This statement declares a union with the tag name `u_example`, which shares memory between a floating-point value `decval`, a pointer to an integer `pnum`, and a double precision floating-point variable `my_value`. The statement also defines one instance of the union with a variable name of `U1`. You can declare further instances of this union with a statement such as this:

```
union u_example U2, U3;
```

Members of a union are accessed in exactly the same way as members of a structure. For example, to assign values to members of `U1` and `U2`, you can write this:

```
U1.decval = 2.5;
U2.decval = 3.5 * U1.decval;
```

TRY IT OUT: USING UNIONS

Here's a simple example that makes use of a union:

```
/* Program 11.8 The operation of a union */
#include <stdio.h>

int main(void)
{
  union u_example
  {
    float decval;
    int pnum;
    double my_value;
  } U1;
```

```
  U1.my_value = 125.5;
  U1.pnum = 10;
  U1.decval = 1000.5f;
  printf("\ndecval = %f   pnum = %d   my_value = %lf",
                     U1.decval, U1.pnum, U1.my_value );

  printf("\nU1 size = %d\ndecval size = %d   pnum size = %d   my_value"
                              " size = %d",sizeof U1, sizeof U1.decval,
                                  sizeof U1.pnum, sizeof U1.my_value);
  return 0;
}
```

How It Works

This example demonstrates the structure and basic operation of a union. You declare the union `U1` as follows:

```
union u_example
{
  float decval;
  int pnum;
  double my_value;
} U1;
```

The three members of the union are of different types and they each require a different amount of storage (assuming your compiler assigns 2 bytes to variables of type `int`).

With the assignment statements, you assign a value to each of the members of the union instance `U1` in turn:

```
  U1.my_value = 125.5;
  U1.pnum = 10;
  U1.decval = 1000.5f;
```

Notice that you reference each member of the union in the same way as you do members of a structure.

The next two statements output each of the three member values, the size of the union `U1`, and the size of each of its members. You get this output (or something close if your machine assigns 4 bytes to variables of type `int`):

```
decval = 1000.500000   pnum = 8192   my_value = 125.50016
U1 size = 8
decval size = 4   pnum size = 2   my_value size = 8
```

The first thing to note is that the last variable that was assigned a value is correct, and the other two have been corrupted. This is to be expected, because they all share the same memory space. The second thing to notice is how little the member `my_value` has been corrupted. This is because only the least significant part of `my_value` is being modified. In a practical situation, such a small error could easily be overlooked, but the ultimate consequences could be dire. You need to take great care when using unions that you aren't using invalid data.

■ **Note** You can see from the output of the sizes of the union and its members that the size of the union is the same as the size of the largest member.

Pointers to Unions

You can also define a pointer to a union with a statement such as this:

```
union u_example *pU;
```

Once the pointer has been defined, you can modify members of the union, via the pointer, with these statements:

```
pU = &U2;
U1.decval = pU->decval;
```

The expression on the right of the second assignment is equivalent to `U2.decval`.

Initializing Unions

If you wish to initialize an instance of a union when you declare it, you can initialize it only with a constant of the same type as the first variable in the union. The union just declared, `u_example`, can be initialized only with a `float` constant, as in the following:

```
union u_example U4 = 3.14f;
```

You can always rearrange the sequence of members in a definition of a union so that the member that you want to initialize occurs first. The sequence of members has no other significance, because all members overlap in the same memory area.

Structures As Union Members

Structures and arrays can be members of a union. It's also possible for a union to be a member of a structure. To illustrate this, you could write the following:

```
struct my_structure
{
  int num1;
  float num2;
  union
  {
    int *pnum;
    float *pfnum;
  } my_U
} samples[5];
```

Here you declare a structure type, `my_structure`, which contains a union without a tag name, so instances of the union can exist only within instances of the structure. This is often described as an **anonymous union**. You also define an array of five instances of the structure, referenced by the variable name `samples`. The union within the structure shares memory between two pointers. To reference members of the union, you use the same notation that you used for nested structures. For example, to access the pointer to `int` in the third element of the structure array, you use the expression appearing on the left in the following statement:

```
samples[2].my_U.pnum = &my_num;
```

You're assuming here that the variable `my_num` has been declared as type `int`.

It's important to realize that when you're using a value stored in a union, you always retrieve the last value assigned. This may seem obvious, but in practice it's all too easy to use a value as `float` that has most recently been stored as an integer, and sometimes the error can be quite subtle, as

shown by the curious output of my_value in Program 11.7. Naturally, you'll usually end up with garbage if you do this. One technique that is often adopted is to embed a union in a struct that also has a member that specifies what type of value is currently stored in the union. For example

```
/* Type code for data in union */
#define TYPE_LONG   1
#define TYPE_FLOAT  2
#define TYPE_CHAR   3

struct Item
{
  int u_type;
  union
  {
    long integer;
    float floating;
    char ch;
  } u;
} var;
```

This defines the Item structure type that contains two members: a value u_type of type int, and an instance u of an anonymous union. The union can store a value of type long, type float or type char, and the u_type member is used to record the type that is currently stored in u.

You could set a value for var like this:

```
var.u.floating = 2.5f;
var.u_type = TYPE_FLOAT;
```

When you are processing var, you need to check what kind of value is stored. Here's an example of how you might do that:

```
switch(var.u_type)
{
  case TYPE_FLOAT:
  printf("\nValue of var is %10f", var.u.floating);
  break;
  case TYPE_LONG:
  printf("\nValue of var is %10ld", var.u.integer);
  break;
  case TYPE_CHAR:
  printf("\nValue of var is %10c", var.u.ch);
  break;
  default:
  printf("\nInvalid union type code.");
  break;
}
```

When working with unions is this way it is usually convenient to put code such as this in a function.

Defining Your Own Data Types

With structures you've come pretty close to defining your own data types. It doesn't look quite right because you must use the keyword struct in your declarations of structure variables. Declaration of a variable for a built-in type is simpler. However, there's a feature of the C language that permits you

to get over this and make the declaration of variables of structure types you've defined follow exactly the same syntax as for the built-in types. You can apply this feature to simplify types derived from the built-in types, but here, with structures, it really comes into its own.

Structures and the typedef Facility

Suppose you have a structure for geometric points with three coordinates, x, y, and z, that you define with the following statement:

```
struct pts
{
  int x;
  int y;
  int z;
};
```

You can now define an alternative name for declaring such structures using the keyword `typedef`. The following statement shows how you might do this:

```
typedef struct pts Point;
```

This statement specifies that the name `Point` is a synonym for `struct pts`.

When you want to declare some instances of the structure `pts`, you can use a statement such as this:

```
Point start_pt;
Point end_pt;
```

Here, you declare the two structure variables `start_pt` and `end_pt`. The `struct` keyword isn't necessary, and you have a very natural way of declaring structure variables. The appearance of the statement is exactly the same form as a declaration for a `float` or an `int`.

You could combine the `typedef` and the structure declaration as follows:

```
typedef struct pts
{
  int x;
  int y;
  int z;
} Point;
```

Don't confuse this with a basic `struct` declaration. Here, `Point` isn't a structure variable name—this is a type name you're defining. When you need to declare structure variables, as you've just seen, you can use a statement such as this:

```
Point my_pt;
```

There's nothing to prevent you from having several types defined that pertain to a single structure type, or any other type for that matter, although this can be confusing in some situations. One application of this that can help to make your program more understandable is where you're using a basic type for a specific kind of value and you would like to use a type name to reflect the kind of variable you're creating. For example, suppose your application involves weights of different kinds, such as weights of components and weights of assembly. You might find it useful to define a type name, `weight`, as a synonym for type `double`. You could do this with the following statement:

```
typedef double weight;
```

You can now declare variables of type `weight`:

```
weight piston = 6.5;
weight valve = 0.35;
```

Of course, these variables are all of type `double` because `weight` is just a synonym for `double`, and you can still declare variables of type `double` in the usual way.

Simplifying Code Using typedef

Another useful application of `typedef` is to simplify complicated types that can arise. Suppose you have occasion to frequently define pointers to the structure `pts`. You could define a type to do this for you with this statement:

```
typedef struct pts *pPoint;
```

Now, when you want to declare some pointers, you can just write this:

```
pPoint pfirst;
pPoint plast;
```

The two variables declared here are both pointers to structures of type `pts`. The declarations are less error-prone to write and they're also easier to understand.

In Chapter 9 I discussed pointers to functions, which are declared with an even more complicated notation. One of the examples has the pointer declaration:

```
int(*pfun)(int, int);                  /* Function pointer declaration */
```

If you are expecting to use several pointers to functions of this kind in a program, you can use `typedef` to declare a generic type for such declarations with this statement:

```
typedef int (*function_pointer)(int, int);    /* Function pointer type */
```

This doesn't declare a variable of type "pointer to a function." This declares `function_pointer` as a type name that you can use to declare a "pointer to function", so you can replace the original declaration of `pfun` with this statement:

```
function_pointer pfun;
```

This is evidently much simpler than what you started with. The benefit in simplicity is even more marked if you have several such pointers to declare, because you can declare three pointers to functions with the following statements:

```
function_pointer pfun1;
function_pointer pfun2;
function_pointer pfun3;
```

Of course, you can also initialize them, so if you assume you have the functions `sum()`, `product()`, and `difference()`, you can declare and initialize your pointers with the following:

```
function_pointer pfun1 = sum;
function_pointer pfun2 = difference;
function_pointer pfun3 = product;
```

The type name that you've defined naturally only applies to "pointers to functions" with the arguments and return type that you specified in the `typedef` statement. If you want something different, you can simply define another type.

Designing a Program

You've reached the end of another long chapter, and it's time to see how you can put what you've learned into practice in the context of a more substantial example.

The Problem

Numerical data is almost always easier and faster to understand when it's presented graphically. The problem that you're going tackle is to write a program that produces a vertical bar chart from a set of data values. This is an interesting problem for a couple of reasons. It will enable you to make use of structures in a practical context. The problem also involves working out how to place and present the bar chart within the space available, which is the kind of messy manipulation that comes up quite often in real-world applications.

The Analysis

You won't be making any assumptions about the size of the "page" that you're going to output to, or the number of columns, or even the scale of the chart. Instead, you'll just write a function that accepts a dimension for the output page and then makes the set of bars fit the page, if possible. This will make the function useful in virtually any situation. You'll store the values in a sequence of structures in a linked list. In this way, you'll just need to pass the first structure to the function, and the function will be able to get at them all. You'll keep the structure very simple, but you can embellish it later with other information of your own design.

Assume that the order in which the bars are to appear in the chart is going to be the same as the order in which the data values are entered, so you won't need to sort them. There will be two functions in your program: a function that generates the bar chart, and a function `main()` that exercises the bar chart generation process.

These are the steps required:

1. Write the bar-chart function.
2. Write a `main()` function to test the bar-chart function once you've written it.

The Solution

This section outlines the steps you'll take to solve the problem.

Step 1

Obviously, you're going to use a structure in this program because that's what this chapter is about. The first stage is to design the structure that you'll use throughout the program. You'll use a `typedef` so that you don't have to keep reusing the keyword `struct`.

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;
```

```
int main(void)
{
  /* Code for main */
}

/* Definition of the bar-chart function */
```

The barTAG structure will define a bar simply by its value. Notice how you define the pointer in the structure to the next structure. This will enable you to store the bars as a linked list, which has the merit that you can allocate memory as you go so none will be wasted. This suits this situation because you'll only ever want to step through the bars from the first to the last. You'll create them in sequence from the input values and append each new bar to the tail of the previous one. You'll then create the visual representation of the bar chart by stepping through the structures in the linked list. You may have thought that the typedef statement would mean that you could use the bar type name that you're defining here. However, you have to use struct barTAG here because at this point the compiler hasn't finished processing the typedef yet, so bar isn't defined. In other words, the barTAG structure is analyzed first by the compiler, after which the typedef can be expedited to define the meaning of bar.

Now you can specify the function prototype for the bar-chart function and put the skeleton of the definition for the function. It will need to have parameters for a pointer to the first bar in the linked list, the page height and width, and the title for the chart to be produced:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40
typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint;     /* Type definition*/

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height, char *title);

int main(void)
{
   /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title)
{
  /* Code for function...  */
  return 0;
}
```

You've added a typedef to define uint as an alternative to unsigned int. This will shorten statements that declare variables of type unsigned int.

Next, you can add some declarations and code for the basic data that you need for the bar chart. You'll need the maximum and minimum values for the bars and the vertical height of the chart, which

will be determined by the difference between the maximum and minimum values. You also need to calculate the width of a bar, given the page width and the number of bars, and you must adjust the height to accommodate a horizontal axis and the title:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint;     /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title);

int main(void)
{
   /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title)
{
  bar *plastbar = pfirstbar;  /* Pointer to previous bar         */
  double max = 0.0;           /* Maximum bar value               */
  double min = 0.0;           /* Minimum bar value               */
  double vert_scale = 0.0;    /* Unit step in vertical direction */
  uint bar_count = 1;         /* Number of bars - at least 1     */
  uint barwidth = 0;          /* Width of a bar                  */
  uint space = 2;             /* spaces between bars             */

   /* Find maximum and minimum of all bar values */

  /* Set max and min to first bar value */
  max = min = plastbar->value;

  while((plastbar = plastbar->pnextbar) != NULL)
  {
    bar_count++;              /* Increment bar count */
    max = (max < plastbar->value)? plastbar->value : max;
    min = (min > plastbar->value)? plastbar->value : min;
  }
  vert_scale = (max - min)/page_height; /* Calculate step length */
```

```
  /* Check bar width */
  if((barwidth = page_width/bar_count - space) < 1)
  {
    printf("\nPage width too narrow.\n");
    return -1;
  }

  /* Code for rest of the function...  */
  return 0;
}
```

The space variable stores the number of spaces separating one bar from the next, and you arbitrarily assign the value 2 for this.

You will, of necessity, be outputting the chart a row at a time. Therefore, you'll need a string that corresponds to a section across a bar that you can use to draw that bar row by row, and a string of the same length, containing spaces to use when there's no bar at a particular position across the page. Let's add the code to create these:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>
#include <stdlib.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint;     /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title);

int main(void)
{
  /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                    char *title)
{
  bar *plastbar = pfirstbar;  /* Pointer to previous bar         */
  double max = 0.0;           /* Maximum bar value               */
  double min = 0.0;           /* Minimum bar value               */
  double vert_scale = 0.0;    /* Unit step in vertical direction */
  uint bar_count = 1;         /* Number of bars - at least 1     */
  uint barwidth = 0;          /* Width of a bar                  */
  uint space = 2;             /* spaces between bars             */
  uint i = 0;                 /* Loop counter                    */
  char *column = NULL;        /* Pointer to bar column section   */
  char *blank = NULL;         /* Blank string for bar+space      */
```

```
  /* Find maximum and minimum of all bar values */

  /* Set max and min to first bar value */
  max = min = plastbar->value;

  while((plastbar = plastbar->pnextbar) != NULL)
  {
    bar_count++;              /* Increment bar count */
    max = (max < plastbar->value)? plastbar->value : max;
    min = (min > plastbar->value)? plastbar->value : min;
  }
  vert_scale = (max - min)/page_height; /* Calculate step length */

  /* Check bar width */
  if((barwidth = page_width/bar_count - space) < 1)
  {
    printf("\nPage width too narrow.\n");
    return -1;
  }

  /* Set up a string that will be used to build the columns */

  /* Get the memory */
  if((column = malloc(barwidth + space + 1)) == NULL)
  {
    printf("\nFailed to allocate memory in barchart()"
                          " - terminating program.\n");
    exit(1);
  }
  for(i = 0 ; i<space ; i++)
    *(column+i)=' ';          /* Blank the space between bars */
  for( ; i<space+barwidth ; i++)
    *(column+i)='#';          /* Enter the bar characters     */
  *(column+i) = '\0';         /* Add string terminator        */

  /* Set up a string that will be used as a blank column */

  /* Get the memory */
  if((blank = malloc(barwidth + space + 1)) == NULL)
  {
    printf("\nFailed to allocate memory in barchart()"
                          " - terminating program.\n");
    exit(1);
  }

  for(i = 0 ; i<space+barwidth ; i++)
    *(blank+i) = ' ';         /* Blank total width of bar+space */
  *(blank+i) = '\0';          /* Add string terminator          */

  /* Code for rest of the function...  */
  free(blank);                /* Free memory for blank string   */
  free(column);               /* Free memory for column string  */
  return 0;
}
```

You'll draw a bar using '#' characters. When you draw a bar, you'll write a string containing space spaces and barwidth '#' characters. You allocate the memory for this dynamically using the library function malloc(), so you must add an #include directive for the header file stdlib.h. The string that you'll use to draw a bar is column, and blank is a string of the same length containing spaces. After the bar chart has been drawn and just before you exit, you free the memory occupied by column and blank.

Next, you can add the final piece of code that draws the chart:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG
{
  double value;
  struct barTAG *pnextbar;
}bar;

typedef unsigned int uint;     /* Type definition */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                  char *title);
int main(void)
{
   /* Code for main */
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height,
                                                  char *title)
{
  bar *plastbar = pfirstbar;  /* Pointer to previous bar            */
  double max = 0.0;           /* Maximum bar value                  */
  double min = 0.0;           /* Minimum bar value                  */
  double vert_scale = 0.0;    /* Unit step in vertical direction    */
  double position = 0.0;      /* Current vertical position on chart */
  uint bar_count = 1;         /* Number of bars - at least 1        */
  uint barwidth = 0;          /* Width of a bar                     */
  uint space = 2;             /* spaces between bars                */
  uint i = 0;                 /* Loop counter                       */
  uint bars = 0;              /* Loop counter through bars          */
  char *column = NULL;        /* Pointer to bar column section      */
  char *blank = NULL;         /* Blank string for bar+space         */
  bool axis = false;          /* Indicates axis drawn               */

  /* Find maximum and minimum of all bar values */

  /* Set max and min to first bar value */
  max = min = plastbar->value;
```

```
while((plastbar = plastbar->pnextbar) != NULL)
{
  bar_count++;              /* Increment bar count */
  max = (max < plastbar->value)? plastbar->value : max;
  min = (min > plastbar->value)? plastbar->value : min;
}
vert_scale = (max - min)/page_height; /* Calculate step length */

/* Check bar width */
if((barwidth = page_width/bar_count - space) < 1)
{
  printf("\nPage width too narrow.\n");
  return -1;
}

/* Set up a string that will be used to build the columns */

/* Get the memory */
if((column = malloc(barwidth + space + 1)) == NULL)
{
  printf("\nFailed to allocate memory in barchart()"
                        " - terminating program.\n");
  exit(1);
}
for(i = 0 ; i<space ; i++)
  *(column+i)=' ';         /* Blank the space between bars */
for( ; i < space+barwidth ; i++)
  *(column+i)='#';         /* Enter the bar characters     */
*(column+i) = '\0';        /* Add string terminator        */

/* Set up a string that will be used as a blank column */

/* Get the memory */
if((blank = malloc(barwidth + space + 1)) == NULL)
{
  printf("\nFailed to allocate memory in barchart()"
                        " - terminating program.\n");
  exit(1);
}

for(i = 0 ; i<space+barwidth ; i++)
  *(blank+i) = ' ';        /* Blank total width of bar+space */
*(blank+i) = '\0';         /* Add string terminator          */

printf("^ %s\n", title);   /* Output the chart title      */
```

```
  /* Draw the bar chart */
  position = max;
  for(i = 0 ; i <= page_height ; i++)
  {
    /* Check if we need to output the horizontal axis */
    if(position <= 0.0 && !axis)
    {
      printf("+");           /* Start of horizontal axis    */
      for(bars = 0; bars < bar_count*(barwidth+space); bars++)
        printf("-");         /* Output horizontal axis      */
      printf(">\n");
      axis = true;           /* Axis was drawn              */
      position -= vert_scale;/* Decrement position           */
      continue;
    }
    printf("|");             /* Output vertical axis        */
    plastbar = pfirstbar;    /* start with the first bar    */

    /* For each bar... */
    for(bars = 1; bars <= bar_count; bars++)
    {
      /* If position is between axis and value, output column */
      /* otherwise output blank                               */
      printf("%s", position <= plastbar->value &&
                   plastbar->value >= 0.0 && position > 0.0 ||
                   position >= plastbar->value &&
                   plastbar->value <= 0.0 &&
                   position <= 0.0 ? column: blank);
      plastbar = plastbar->pnextbar;
    }
    printf("\n");            /* End the line of output      */
    position -= vert_scale;  /* Decrement position          */
  }
  if(!axis)            /* Have we output the horizontal axis? */
  {                    /* No, so do it now                    */
    printf("+");
    for(bars = 0; bars < bar_count*(barwidth+space); bars++)
      printf("-");
    printf(">\n");
  }

  free(blank);               /* Free memory for blank string   */
  free(column);              /* Free memory for column string  */
  return 0;
}
```

The `for` loop outputs `page_height` lines of characters. Each line will represent a distance of `vert_scale` on the vertical axis. You get this value by dividing `page_height` by the difference between the maximum and minimum values. Therefore, the first line of output corresponds to `position` having the value `max`, and it's decremented by `vert_scale` on each iteration until it reaches `min`.

On each line, you must decide first if you need to output the horizontal axis. This will be necessary when `position` is less than or equal to 0 and you haven't already displayed the axis.

On lines other than the horizontal axis, you must decide what to display for each bar position. This is done in the inner `for` loop that repeats for each bar. The conditional operator in the `printf()` call outputs either `column` or `blank`. You output `column` if `position` is between the value of the bar and 0, and you output `blank` otherwise. Having output a complete row of bar segments, you output `'\n'` to end the line and decrement the value of `position`.

It's possible that all the bars could be positive, in which case you need to make sure that the horizontal axis is output after the loop is complete, because it won't be output from within the loop.

Step 2

Now you just need to implement `main()` to exercise the `bar_chart()` function:

```
/* Program 11.9 Generating a bar chart */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define PAGE_HEIGHT  20
#define PAGE_WIDTH   40

typedef struct barTAG         /* Bar structure       */
{
  double value;              /* Value of bar        */
  struct barTAG *pnextbar;   /* Pointer to next bar */
}bar;                         /* Type for a bar      */

typedef unsigned int uint;    /* Type definition     */

/* Function prototype */
int bar_chart(bar *pfirstbar, uint page_width, uint page_height, char *title);

int main()
{
  bar firstbar;             /* First bar structure */
  bar *plastbar = NULL;     /* Pointer to last bar */
  char value[80];           /* Input buffer        */
  char title[80];           /* Chart title         */

  printf("\nEnter the chart title: ");
  gets(title);              /* Read chart title    */

  for( ;; )                 /* Loop for bar input  */
  {
    printf("Enter the value of the bar, or use quit to end: ");
    gets(value);
```

```
    if(strcmp(value, "quit") == 0)   /* quit entered?       */
     break;                          /* then input finished */

    /* Store in next bar */
    if(!plastbar)                   /* First time?         */
    {
      firstbar.pnextbar = NULL;  /* Initialize next pointer */
      plastbar = &firstbar;      /* Use the first           */
    }
    else
    {
      /* Get memory */
      if(!(plastbar-> = malloc(sizeof(bar))))
      {
        printf("Oops! Couldn't allocate memory\n");
        return -1;
      }
      plastbar = plastbar->pnextbar;    /* Old next is new bar  */
      plastbar->pnextbar = NULL;        /* New bar next is NULL */
    }
    plastbar->value = atof(value);      /* Store the value      */
  }

  /* Create bar-chart */
  bar_chart(&firstbar, PAGE_WIDTH, PAGE_HEIGHT, title);

  /* We are done, so release all the memory we allocated */
  while(firstbar.pnextbar)
  {
    plastbar = firstbar.pnextbar;           /* Save pointer to next */
    firstbar.pnextbar = plastbar->pnextbar; /* Get one after next   */
    free(plastbar);                         /* Free next memory     */
  }
  return 0;
}

int bar_chart(bar *pfirstbar, uint page_width, uint page_height, char *title)
{
  /* Implementation of function as before... */
}
```

After reading the chart title using `gets()`, you read successive values in the `for` loop. For each value other than the first, you allocate the memory for a new bar structure before storing the value. Of course, you keep track of the first structure, `firstbar`, because this is the link to all the others, and you track the pointer to the last structure that you added so that you can update its `pnextbar` pointer when you add another. Once you have all the values, you call `bar_chart()` to produce the chart. Finally, you delete the memory for the bars. Note that you need to take care to not delete `firstbar`, as you didn't allocate the memory for this dynamically. You need an `#include` directive for `string.h` because you use the `gets()` function.

All you do then is add a line to `main()` that actually prints the chart from the values typed in. Typical output from the example is shown here:

```
Enter the chart title: Trial Bar Chart
Enter the value of the bar, or use quit to end: 6
Enter the value of the bar, or use quit to end: 3
Enter the value of the bar, or use quit to end: -5
Enter the value of the bar, or use quit to end: -7
Enter the value of the bar, or use quit to end: 9
Enter the value of the bar, or use quit to end: 4
Enter the value of the bar, or use quit to end: quit

^ Trial Bar Chart
|                          ####
|                          ####
|                          ####
|                          ####
|  ####                    ####
|  ####                    ####
|  ####                    ####
|  ####                    ####  ####
|  ####  ####              ####  ####
|  ####  ####              ####  ####
|  ####  ####              ####  ####
|  ####  ####              ####  ####
+----------------------------------->
|              ####  ####
|              ####  ####
|              ####  ####
|              ####  ####
|              ####  ####
|                    ####
|                    ####
|                    ####
```

Summary

This has been something of a marathon chapter, but the topic is extremely important. Having a good grasp of structures rates alongside understanding pointers and functions in importance, if you want to use C effectively.

Most real-world applications deal with things such as people, cars, or materials, which require several different values to represent them. Structures in C provide a ready tool for dealing with these sorts of complex objects. Although some of the operations may seem a little complicated, remember that you're dealing with complicated entities, so the complexity isn't implicit in the programming capability; rather, it's built into the problem you're tackling.

In the next chapter, you'll look at how you can store data in external files. This will, of course, include the ability to store structures.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 11-1. Define a `struct` type with the name `Length` that represents a length in yards, feet, and inches. Define an `add()` function that will add two `Length` arguments and return the sum as type `Length`. Define a second function, `show()`, that will display the value of its `Length` argument. Write a program that will use the `Length` type and the `add()` and `show()` functions to sum an arbitrary number of lengths in yards, feet, and inches that are entered from the keyboard and output the total length.

Exercise 11-2. Define a `struct` type that contains a person's name consisting of a first name and a second name, plus the person's phone number. Use this `struct` in a program that will allow one or more names and corresponding numbers to be entered and will store the entries in an array of structures. The program should allow a second name to be entered and output all the numbers corresponding to the name, and optionally output all the names with their corresponding numbers.

Exercise 11-3. Modify or reimplement the program from the previous exercise to store the structures in a linked list in ascending alphabetical order of the names.

Exercise 11-4. Write a program to use a struct to count the number of occurrences of each different word in a paragraph of text that's entered from the keyboard.

Exercise 11-5. Write a program that reads an arbitrary number of names consisting of a first name followed by a last name. The program should use a binary tree to output the names in ascending alphabetical sequence ordered by first name within second name (i.e., second name takes precedence in the ordering so Ann Choosy comes after Bill Champ and before Arthur Choosy).

CHAPTER 12

Working with Files

If your computer could only ever process data stored within the main memory of the machine, the scope and variety of applications that you could deal with would be severely limited. Virtually all serious business applications require more data than would fit into main memory and depend on the ability to process data that's stored on an external device, such as a fixed disk drive. In this chapter, you'll explore how you can process data stored in files on an external device.

C provides a range of functions in the header file `<stdio.h>` for writing to and reading from external devices. The external device you would use for storing and retrieving data is typically a fixed disk drive, but not exclusively. Because, consistent with the philosophy of the C language, the library facilities that you'll use for working with files are device-independent, so they apply to virtually any external storage device. However, I'll assume in the examples in this chapter that we are dealing with disk files.

In this chapter you'll learn the following:

- What a file is in C
- How files are processed
- How to write and read formatted files and binary files
- How to retrieve data from a file by direct random access to the information
- How to use temporary work files in a program
- How to update binary files
- How to write a file viewer program

The Concept of a File

With all the examples you've seen up to now, any data that the user enters when the program is executed is lost once the program finishes running. At the moment, if the user wants to run the program with the same data, he or she must enter it again each time. There are a lot of occasions when this is not only inconvenient, but also makes the programming task impossible.

If you want to maintain a directory of names, addresses, and telephone numbers, for instance, a program in which you have to enter all the names, addresses, and telephone numbers each time you run it is worse than useless! The answer is to store data on permanent storage that continues to be maintained after your computer is switched off. As I'm sure you know, this storage is called a **file**, and a file is usually stored on a hard disk.

You're probably familiar with the basic mechanics of how a disk works. If so, this can help you recognize when a particular approach to file usage is efficient and when it isn't. On the other hand, if you know nothing about disk file mechanics, don't worry at this point. There's nothing in the concept of file processing in C that depends on any knowledge of physical storage devices.

A file is essentially a serial sequence of bytes, as illustrated in Figure 12-1.

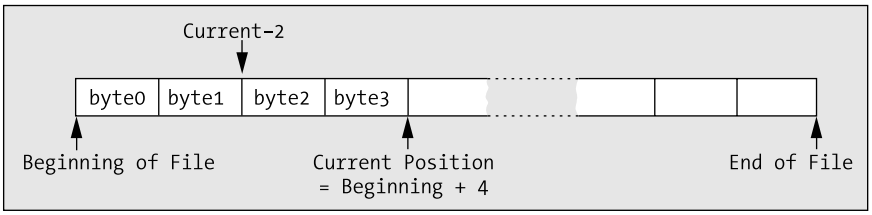


Figure 12-1. *Structure of a file*

Positions in a File

A file has a beginning and an end, and it has a **current position**, typically defined as so many bytes from the beginning, as Figure 12-1 illustrates. The current position is where any file action (a read from the file or a write to the file) will take place. You can move the current position to any other point in the file. A new current position can be specified as an offset from the beginning of the file or, in some circumstances, as a positive or negative offset from the previous current position.

File Streams

The C library provides functions for reading and writing to or from data streams. A **stream** is an abstract representation of any external source or destination for data, so the keyboard, the command line on your display, and files on disk are all examples of streams. You therefore use the same input/output functions for reading and writing any external device that is mapped to a stream.

There are two ways of writing data to a stream that is a disk file. Firstly, you can write a file as a **text file**, in which case data is written as a characters organized as lines, where each line is terminated by a newline character. Obviously, binary data such as values of type `int` or type `double` have to be converted to characters to allow them to be written to a text file, and you've already seen how this formatting is done with the `printf()` function. Secondly you can write a file as a **binary file**. Data that is written to a binary file is always written as a series of bytes, exactly as it appears in memory, so a value of type `double` for example would be written as the 8 bytes that appear in memory.

Of course, you can write any data you like to a file, but once a file has been written, it just consists of a series of bytes on disk. Regardless of whether you write a file as a binary file or as a text file, it ultimately ends up as just a series of bytes, whatever the data is. This means that when the file is read, the program must know what sort of data the file represents. You've seen many times now that exactly what a series of bytes represents is dependent upon how you interpret it. A sequence of 12 bytes in a binary file could be 12 characters, 12 8-bit signed integers, 12 8-bit unsigned integers, 6 16-bit signed integers, a 32-bit integer followed by an 8-byte floating-point value, and so on. All of these will be more or less valid interpretations of the data, so it's important that a program that is reading a file has the correct assumptions about how it was written.

Accessing Files

The files that are resident on your disk drive each have a name, and the rules for naming files will be determined by your operating system. When you write a program to process a file, it would not be particularly convenient if the program would only work with a specific file with a particular name. If it did, you would need to produce a different program for each file you might want to process. For this reason, when you process a file in C, your program references a file through a **file pointer**. A file

pointer is an abstract pointer that is associated with a particular file when the program is run so that the program can work with different files on different occasions. A file pointer points to a `struct` that represents a stream. In the examples in this chapter, I'll use Microsoft Windows file names. If you're using a different operating system environment, such as UNIX, you'll need to adjust the names of the files appropriately.

If you want to use several files simultaneously in a program, you need a separate file pointer for each file, although as soon as you've finished using one file, you can associate the file pointer you were using with another file. So if you need to process several files, but you'll be working with them one at a time, you can do it with one file pointer.

Opening a File

You associate a specific external file name with an internal file pointer variable through a process referred to as **opening a file**. You open a file by calling the standard library function `fopen()`, which returns the file pointer for a specific external file. The function `fopen()` is defined in `<stdio.h>`, and it has this prototype:

```
FILE *fopen(char *name, char *mode);
```

The first argument to the function is a pointer to a string that is the name of the external file that you want to process. You can specify the name explicitly as an argument, or you can use an array, or a variable of type pointer to `char` that contains the address of the character string that defines the file name. You would typically obtain the file name through some external means, such as from the command line when the program is started, or you could arrange to read it in from the keyboard. Of course, you can also define a file name as a constant at the beginning of a program when the program always works with the same file.

The second argument to the `fopen()`function is a character string called the **file mode** that specifies what you want to do with the file. As you'll see, this spans a whole range of possibilities, but for the moment I'll introduce just three file modes (which nonetheless comprise the basic set of operations on a file). Table 12-1 lists these three file modes.

Table 12-1. *File Modes*

Mode	Description
`"w"`	Open a text file for **w**rite operations. If the file exists, its current contents are discarded.
`"a"`	Open a text file for **a**ppend operations. All writes are to the end of the file.
`"r"`	Open a text file for **r**ead operations.

■ **Note** Notice that a file mode specification is a character string between double quotes, not a single character between single quotes.

These three modes only apply to text files that are files that are written as characters. You can also work with binary files that are written as a sequence of bytes and I'll discuss that in the section "Binary File Input and Output" later in this chapter. Assuming the call to `fopen()` is successful, the function returns a pointer of type `File *` that you can use to reference the file in further input/output

operations, using other functions in the library. If the file cannot be opened for some reason, fopen() returns a null pointer.

■ **Note** The pointer returned by fopen() is referred to as a **file pointer**, or a **stream pointer**.

So a call to fopen() does two things for you: it creates a file pointer that identifies the specific file on disk that your program is going to operate on, and it determines what you can do with that file within your program.

The pointer that's returned by fopen() is of type FILE * or "pointer to FILE," where FILE specifies a structure type that has been predefined in the header file <stdio.h> through a typedef. The structure that a file pointer points to will contain information about the file. This will be such things as the open mode you specified, the address of the buffer in memory to be used for data, and a pointer to the current position in the file for the next operation. You don't need to worry about the contents of this structure in practice. It's all taken care of by the input/output functions. However, if you really want to know about the FILE structure, you can browse through the library header file.

As I mentioned earlier, when you want to have several files open at once, they must each have their own file pointer variable declared, and you open each of them with a separate call to fopen () with the value that is returned stored in a separate file pointer. There's a limit to the number of files you can have open at one time that will be determined by the value of the constant FOPEN_MAX that's defined in <stdio.h>. FOPEN_MAX is an integer that specifies the maximum number of streams that can be open at one time. The C language standard requires that the value of FOPEN_MAX be at least 8, including stdin, stdout and stderr. Thus, as a minimum, you will be able to be working with up to 5 files simultaneously.

If you want to write to an existing text file with the name myfile.txt, you would use these statements:

```
FILE *pfile = fopen("myfile.txt", "w");  /* Open file myfile.txt to write it */
```

This statement opens the file and associates the physical file specified by the file name myfile.txt with your internal pointer pfile. Because you've specified the mode as "w", you can only write to the file; you can't read from it. The string that you supply as the first argument is limited to a maximum of FILENAME_MAX characters, where FILENAME_MAX is defined in the <stdio.h> header file. This value is usually sufficiently large enough that it isn't a real restriction.

If a file with the name myfile.txt does not already exist, the call to the function fopen() in the previous statement will create a new file with this name. Because you have just provided the file name without any path specification as the first argument to the fopen() function, the file is assumed to be in the current directory, and if the file is not found there, that's where it will be created. You can also specify a string that is the full path and name for the file, in which case the file will be assumed to be at that location and a new file will be created there if necessary. Note that if the directory that's supposed to contain the file doesn't exist when you specify the file path, neither the directory nor the file will be created and the fopen() call will fail. If the call to fopen() does fail for any reason, NULL will be returned. If you then attempt further operations with a NULL file pointer, it will cause your program to terminate.

■ **Note** So here you have the facility to create a new text file. Simply call fopen() with mode "w" and the first argument specifying the name you want to assign to the new file.

On opening a file for writing, the file is positioned at the beginning of any existing data for the first operation. This means that any data that was previously written to the file will be overwritten when you initiate any write operations.

If you want to add to an existing text file rather than overwrite it, you specify mode `"a"`, which is the append mode of operation. This positions the file at the end of any previously written data. If the file specified doesn't exist, as in the case of mode `"w"`, a new file will be created. Using the file pointer that you declared previously, to open the file to add data to the end, use the following statement:

```
pfile = fopen("myfile.txt", "a");      /* Open file myfile.txt to add to it */
```

When you open a file in append mode, all write operations will be at the end of the data in the file on each write operation. In other words, all write operations append data to the file and you cannot update the existing contents in this mode.

If you want to read a file, once you've declared your file pointer, open it using this statement:

```
pfile = fopen("myfile.txt", "r");
```

Because you've specified the mode argument as `"r"`, indicating that you want to read the file, you can't write to this file. The file position will be set to the beginning of the data in the file.

Clearly, if you're going to read the file, it must already exist. If you inadvertently try to open a file for reading that doesn't exist, `fopen()` will return `NULL`. It's therefore a good idea to check the value returned from `fopen()` in an `if` statement, to make sure that you really are accessing the file you want.

Renaming a File

There are many circumstances in which you'll want to rename a file. You might be updating the contents of a file by writing a new, updated file, for instance. You'll probably want to assign a temporary name to the new file while you're creating it, and then change the name to that of the old file once you've deleted it. Renaming a file is very easy. You just use the `rename()` function, which has the following prototype:

```
int rename(const char *oldname, const char *newname);
```

The integer that's returned will be 0 if the name change is successful, and nonzero otherwise. The file must be closed when you call `rename()`, otherwise the operation will fail.

Here's an example of using the `rename()` function:

```
if(rename( "C:\\temp\\myfile.txt", "C:\\temp\\myfile_copy.txt"))
  printf("Failed to rename file.");
else
  printf("File renamed successfully.");
```

The preceding code fragment will change the name of the `myfile.txt` file in the `temp` directory on drive C to `myfile_copy.text`. A message will be produced that indicates whether the name change succeeded. Obviously, if the file path is incorrect or the file doesn't exist, the renaming operation will fail.

■ **Caution** Note the double backslash in the file path string. If you forget to use the escape sequence for a backslash when specifying a Microsoft Windows file path you won't get the file name that you want.

Closing a File

When you've finished with a file, you need to tell the operating system that this is the case and free up your file pointer. This is referred to as **closing** a file. You do this by calling the `fclose()`function which accepts a file pointer as an argument and returns a value of type `int`, which will be `EOF` if an error occurs and 0 otherwise. The typical usage of the `fclose()` function is as follows:

```
fclose(pfile);                          /* Close the file associated with pfile */
```

The result of executing this statement is that the connection between the pointer, `pfile`, and the physical file name is broken, so `pfile` can no longer be used to access the physical file it represented. If the file was being written, the current contents of the output buffer are written to the file to ensure that data isn't lost.

■ **Note** `EOF` is a special character called the **end-of-file character**. In fact, the symbol `EOF` is defined in `<stdio.h>` and is usually equivalent to the value –1. However, this isn't necessarily always the case, so you should use `EOF` in your programs rather than an explicit value. EOF generally indicates that no more data is available from a stream.

It's good programming practice to close a file as soon as you've finished with it. This protects against output data loss, which could occur if an error in another part of your program caused the execution to be stopped in an abnormal fashion. This could result in the contents of the output buffer being lost, as the file wouldn't be closed properly. You must also close a file before attempting to rename it or remove it.

■ **Note** Another reason for closing files as soon as you've finished with them is that the operating system will usually limit the number of files you may have open at one time. Closing files as soon as you've finished with them minimizes the chances of you falling afoul of the operating system in this respect.

There is a function in `<stdio.h>` that will force any unwritten data left in a buffer to be written to a file. This is the function `fflush()`, which you've already used in previous chapters to flush the input buffer. With your file pointer `pfile`, you could force any data left in the output buffer to be written to the file by using this statement:

```
fflush(pfile);
```

The `fflush()` function returns a value of type `int`, which is normally 0 but will be set to `EOF` if an error occurs.

Deleting a File

Because you have the ability to create a file in your code, at some point you'll want to be able to delete a file programmatically, too. The `remove()` function that's declared in `<stdio.h>` does this. You use it like this:

```
remove("pfile.txt");
```

This will delete the file from the current directory that has the name `pfile.txt`. Note that the file should not be open when you call `remove()` to delete it. If the file is open, the effect of calling remove is implementation-defined, so consult your library documentation.

You always need to double-check any operations on files, but you need to take particular care with operations that delete files.

Writing to a Text File

Once you've opened a file for writing, you can write to it any time from anywhere in your program, provided you have access to the pointer for the file that has been set by `fopen()`. So if you want to be able to access a file from anywhere in a program that contains multiple functions, you need to ensure the file pointer has global scope or arrange for it to be passed as an argument to any function that accesses the file.

■ **Note** As you'll recall, to ensure that the file pointer has global scope you place the declaration for it outside of all of the functions, usually at the beginning of the source file.

The simplest write operation is provided by the function `fputc()`, which writes a single character to a text file. It has the following prototype:

```
int fputc(int c, FILE *pfile);
```

The `fputc()` function writes the character specified by the first argument to the file defined by the second argument, which is a file pointer. If the write is successful, it returns the character that was written. Otherwise it returns `EOF`.

In practice, characters aren't written to the physical file one by one. This would be extremely inefficient. Hidden from your program and managed by the output routine, output characters are written to an area of memory called a **buffer** until a reasonable number have been accumulated; they are then all written to the file in one go. This mechanism is illustrated in Figure 12-2.

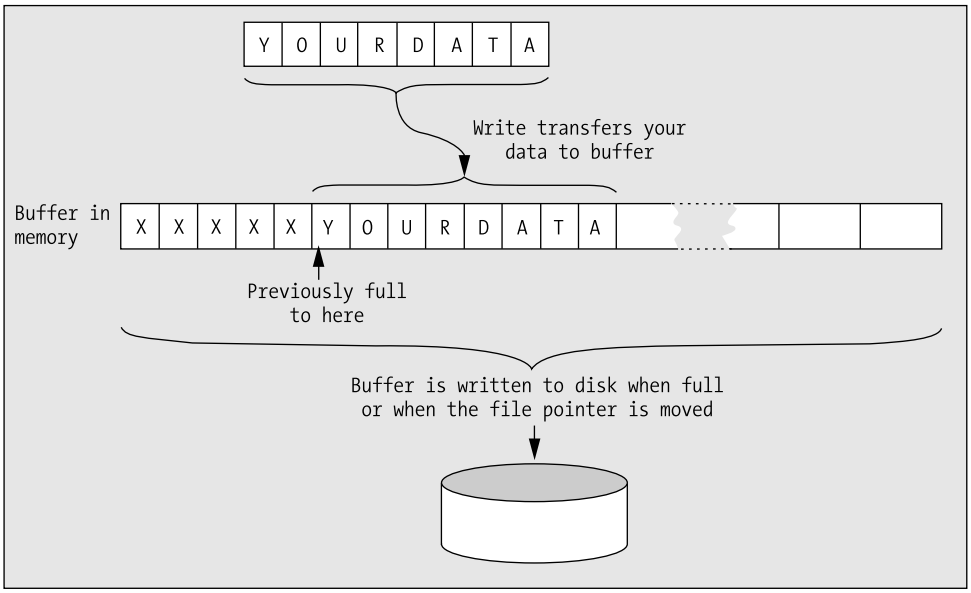


Figure 12-2. *Writing a file*

Note that the putc()function is equivalent to fputc(). It requires the same arguments and the return type is the same. The difference between them is that putc() may be implemented in the standard library as a macro, whereas fputc() is definitely a function.

Reading from a Text File

The fgetc() function is complementary to fputc() and reads a character from a text file that has been opened for reading. It takes a file pointer as its only argument and returns the character read as type int if the read is successful; otherwise, it returns EOF. The typical use of fgetc() is illustrated by the following statement:

```
mchar = fgetc(pfile);                    /* Reads a character into mchar */
```

You're assuming here that the variable mchar has been declared to be of type int.

Behind the scenes, the actual mechanism for reading a file is the inverse of writing to a file. A whole block of characters is read into a buffer in one go. The characters are then handed over to your program one at a time as you request them, until the buffer is empty, whereupon another block is read. This makes the process very fast, because most fgetc() operations won't involve reading the disk but simply moving a character from the buffer in main memory to the place where you want to store it.

Note that the function getc() that's equivalent to fgetc() is also available. It requires an argument of type FILE* and returns the character read as type int, so it's virtually identical to fgetc(). The only difference between them is that getc() may be implemented as a macro, whereas fgetc() is a function.

■ **Caution** Don't confuse the function getc() with the function gets(). They're quite different in operation: getc() reads a single character from the stream specified by its argument, whereas gets() reads a whole line of input from the standard input stream, which is the keyboard. You've already used the gets() function in previous chapters for reading a string from the keyboard.

TRY IT OUT: USING A SIMPLE FILE

You now have enough knowledge of the file input/output capabilities in C to write a simple program that writes a file and then reads it. So let's do just that:

```
/* Program 12.1 Writing a file a character at a time */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

const int LENGTH = 80;                /* Maximum input length   */

int main(void)
{
  char mystr[LENGTH];                 /* Input string           */
  int lstr = 0;                       /* Length of input string */
  int mychar = 0;                     /* Character for output   */
  FILE *pfile = NULL;                 /* File pointer           */
  char *filename = "C:\\myfile.txt";
```

```
  printf("\nEnter an interesting string of less than 80 characters:\n");
  fgets(mystr, LENGTH, stdin);        /* Read in a string       */

  /* Create a new file we can write */

  if(!(pfile = fopen(filename, "w")))
  {
    printf("Error opening %s for writing. Program terminated.", filename);
    exit(1);
  }

  lstr = strlen(mystr);
  for(int i = lstr-1 ; i >= 0 ; i--)
    fputc(mystr[i], pfile);           /* Write string to file backward  */

  fclose(pfile);                      /* Close the file                 */

  /* Open the file for reading */
  if(!(pfile = fopen(filename, "r")))
{
    printf("Error opening %s for reading. Program terminated.", filename);
    exit(1);
  }

  /* Read a character from the file and display it */
  while((mychar = fgetc(pfile)) != EOF)
    putchar(mychar);                  /* Output character from the file */
  putchar('\n');                      /* Write newline                  */

  fclose(pfile);                      /* Close the file                 */
  remove(filename);                   /* Delete the physical file       */
  return 0;
}
```

Here's an example of some output from this program:

```
Enter an interesting string.
Too many cooks spoil the broth.
.htorb eht liops skooc ynam ooT
```

How It Works

The name of the file that you're going to work with is defined by this statement:

```
char *filename = "C:\\myfile.txt";
```

This statement defines the file with the name `myfile.txt` on drive C with the Microsoft Windows notation for file names. As I noted earlier, you must use the escape sequence `'\\'` to get a backslash character. If you forget to do this and just use a single backslash, the compiler will think that you're writing an escape sequence `'\m'` in this case, which it won't recognize as valid.

Before running this program—or indeed any of the examples working with files—do make sure you don't have an existing file with the same name and path. If you have a file with the same name as that used in the example, you should change the initial value for `filename` in the example; otherwise, your existing file will be overwritten.

After displaying a prompt, the program reads a string from the keyboard. It then executes the following statements:

```
if(!(pfile = fopen(filename, "w")))
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit (1);
}
```

The condition in this `if` statement calls `fopen()` to create the new file `myfile.txt` on drive C, opens it for writing, and stores the pointer that is returned in `pfile`. The second argument to `fopen()` determines the mode as writing the file. The block of statements will be executed if `fopen()` returns `NULL`, so in this case you display a message and call the `exit()` function that is declared in `<stdlib.h>` for an abnormal end to the program.

After determining the `length` of the string using `strlen()` and storing the result in `lstr`, you have a loop defined by these statements:

```
for(int i = lstr-1 ; i >= 0 ; i--)
  fputc(mystr[i], pfile);         /* Write string to file backward  */
```

The loop index is varied from a value corresponding to the last character in the string `lstr-1` back to 0. Therefore, the `putc()` function call within the loop writes to the new file character-by-character, in reverse order. The particular file you're writing is specified by the pointer `pfile` as the second argument to the function call.

After closing the file with a call to `fclose()`, it's reopened in reading mode by these statements:

```
if(!(pfile = fopen(filename, "r")))
{
  printf("Error opening %s for reading. Program terminated.", filename);
  exit(1);
}
```

The mode specification `"r"` indicates that you intend to read the file, so the file position will be set to the beginning of the file. You have the same check for a `NULL` return value as when you wrote the file.

Next, you use the `getc()` function to read characters from the file within the `while` loop condition:

```
while((mychar = fgetc(pfile)) != EOF)
  putchar(mychar);                /* Output character from the file */
```

The file is read character-by-character. The read operation actually takes place within the loop continuation condition. As each character is read, it's displayed on the screen using the function `putc()` within the loop. The process stops when `EOF` is returned by `getc()` at the end of the file.

The last two statements before the return in the `main()` function are the following:

```
fclose(pfile);                    /* Close the file               */
remove(filename);                 /* Delete the physical file     */
```

These statements provide the necessary final tidying up, now that you've finished with the file. After closing the file, the program calls the `remove()` function, which will delete the file identified by the argument. This avoids cluttering up the disk with stray files. If you want to check the contents of the file that was written using a text editor, just remove or comment out the call to `remove()`.

Writing Strings to a Text File

Analogous to the `puts()` function for writing a string to `stdout`, you have the `fputs()` function for writing a string to a text file. Its prototype is as follows:

```
int fputs(char *pstr, FILE *pfile);
```

The first argument is a pointer to the character string that's to be written to the file, and the second argument is a file pointer. The operation of the function is slightly odd, in that it continues to write characters from a string until it reaches a '\0' character, which it doesn't write to the file. This can complicate reading back variable-length strings from a file that have been written by fputs(). It works this way because it's a character write operation, not a binary write operation, so it's expecting to write a line of text that has a newline character at the end. A newline character isn't required by the operation of the function, but it's very helpful when you want to read the file back (using the complementary fgets() function, as you'll see).

The fputs() function returns EOF if an error occurs, and 0 under normal circumstances. You use it in the same way as puts(), for example

```
fputs("The higher the fewer", pfile);
```

This will output the string appearing as the first argument to the file pointed to by pfile.

Reading Strings from a Text File

Complementing fputs() is the function fgets() for reading a string from a text file. It has the following prototype:

```
char *fgets(char *pstr, int nchars, FILE *pfile);
```

The fgets() function has three parameters. The function will read a string into the memory area pointed to by pstr, from the file specified by pfile. Characters are read from the file until either a '\n' is read or nchars-1 characters have been read from the file, whichever occurs first.

If a newline character is read, it's retained in the string. A '\0' character will be appended to the end of the string in any event. If there is no error, fgets() will return the pointer, pstr; otherwise, NULL is returned. The second argument to this function enables you to ensure that you don't overrun the memory area that you've assigned for input in your program. To prevent the capacity of your data input area from being exceeded, just specify the length of the area or the array that will receive the input data as the second argument to the function.

TRY IT OUT: TRANSFERRING STRINGS TO AND FROM A TEXT FILE

You can exercise the functions to transfer strings to and from a text file in an example that also uses the append mode for writing a file:

```
/* Program 12.2  As the saying goes...it comes back! */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

const int LENGTH = 80;                  /* Maximum input length */

int main(void)
{
  char *proverbs[] =
```

```
           {  "Many a mickle makes a muckle.\n",
              "Too many cooks spoil the broth.\n",
              "He who laughs last didn't get the joke in"
                                  " the first place.\n"
           };

char more[LENGTH];                  /* Stores a new proverb */
FILE *pfile = NULL;                 /* File pointer         */
char *filename = "C:\\myfile.txt";

/* Create a new file( if myfile.txt does not exist */
if(!(pfile = fopen(filename, "w")))    /* Open the file to write it */
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit(1);
}

/* Write our first three sayings to the file. */
int count = sizeof proverbs/sizeof proverbs[0];
for(int i = 0 ; i < count ; i++)
  fputs(proverbs[i], pfile);

fclose(pfile);                      /* Close the file */

/* Open the file to append more proverbs */
if(!(pfile = fopen(filename, "a")))
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit(1);
}

printf("Enter proverbs of less than 80 characters or press Enter to end:\n");
while(true)
{
  fgets(more, LENGTH, stdin);          /* Read a proverb           */
  if(more[0] == '\n')                  /* If its empty line        */
    break;                             /* end input operation      */
  fputs(more, pfile);                  /* Write the new proverb    */
}

fclose(pfile);                         /* Close the file           */

if(!(pfile = fopen(filename, "r")))    /* Open the file to read it */
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit(1);
}

/* Read and output the file contents */
printf("The proverbs in the file are:\n\n");
while(fgets(more, LENGTH, pfile))      /* Read a proverb */
 printf("%s", more);                   /* and display it */
```

```
  fclose(pfile);                          /* Close the file */
  remove(filename);                       /* and remove it  */
  return 0;
}
```

Here is some sample output from this program:

```
Enter proverbs of less than 80 characters or press Enter to end:
Least said, soonest mended.
A nod is as good as a wink to a blind horse.

The proverbs in the file are:

Many a mickle makes a muckle.
Too many cooks spoil the broth.
He who laughs last didn't get the joke in the first place.
Least said, soonest mended.
A nod is as good as a wink to a blind horse.
```

How It Works

You initialize the array of pointers, `proverbs[]`, in the following statement:

```
char *proverbs[] =
           {  "Many a mickle makes a muckle.\n",
              "Too many cooks spoil the broth.\n",
              "He who laughs last didn't get the joke in"
                                  " the first place.\n"
           };
```

You specify the three sayings as initial values for the array elements, and this causes the compiler to allocate the space necessary to store each string.

You have a further declaration of an array that will store a proverb that will be read from the keyboard:

```
char more[LENGTH];                    /* Stores a new proverb */
```

This initializes a conventional `char` array with another proverb. You also include `'\n'` at the end for the same reason as before.

After creating and opening a file on drive C for writing, the program writes the initial three proverbs to the file in a loop:

```
int count = sizeof proverbs/sizeof proverbs[0];
for(int i = 0 ; i < count ; i++)
  fputs(proverbs[i], pfile);
```

The contents of each of the memory areas pointed to by elements of the `proverbs[]` array are written to the file in the `for` loop using the function `fputs()`. This function is extremely easy to use; it just requires a pointer to the string as the first argument and a pointer to the file as the second.

The number of proverbs in the array is calculated by the following expression:

```
sizeof proverbs/sizeof proverbs[0]
```

The expression `sizeof proverbs` will evaluate to the total number of bytes occupied by the complete array, and `sizeof proverbs[0]` will result in the number of bytes required to store a single pointer in one element of the array. Therefore, the whole expression will evaluate to the number of elements in the pointer array. You could

have manually counted how many initializing strings you supplied, of course, but doing it this way means that the correct number of iterations is determined automatically, and this expression will still be correct even if the array dimension is changed by adding more initializing strings.

Once the first set of proverbs has been written, the file is closed and then reopened with this statement:

```
if(!(pfile = fopen(filename, "a")))
{
  printf("Error opening %s for writing. Program terminated.", filename);
  exit(1);
}
```

Because you have the mode specified as `"a"`, the file is opened in append mode. Note that the current position for the file is automatically set to the end of the file in this mode, so that subsequent write operations will be appended to the end of the existing data in the file.

After prompting for input, you read more proverbs from the keyboard and write them to the file with the following statements:

```
while(true)
{
  fgets(more, LENGTH, stdin);          /* Read a proverb           */
  if(more[0] == '\n')                  /* If its empty line        */
    break;                             /* end input operation      */
  fputs(more, pfile);                  /* Write the new proverb    */
}
```

Each additional proverb that's stored in the `more` array is written to the file using `fputs()`. As you can see, the function `fputs()` is just as easy to use with an array as it is with a pointer. Because you're in append mode, each new proverb will be added at the end of the existing data in the file. The loop terminates when an empty line is entered. An empty line will result in a string containing just `'\n'` followed by the string terminator.

Having written the file, you close it and then reopen it for reading, using the mode specifier `"r"`. You then have the following loop:

```
while(fgets(more, LENGTH, pfile))      /* Read a proverb */
 printf("%s", more);                   /* and display it */
```

You read strings successively from the file into the `more` array within the loop continuation condition. After each string is read, you display it on the screen by the call to `printf()` within the loop. The reading of each proverb by `fgets()` is terminated by detecting the `'\n'` character at the end of each string. The loop terminates when the function `fgets()` returns `NULL`.

Finally, the file is closed and then deleted using the function `remove()` in the same fashion as the previous example.

Formatted File Input and Output

Writing characters and strings to a text file is all very well as far as it goes, but you normally have many other types of data in your programs. To write numerical data to a text file, you need something more than you've seen so far, and where the contents of a file are to be human readable, you need a character representation of the numerical data. The mechanism for doing just this is provided by the functions for formatted file input and output.

Formatted Output to a File

You already encountered the function for formatted output to a file when I discussed standard streams back in Chapter 10. It's virtually the same as the `printf()` statement, except that there's one extra parameter and a slight name change. Its typical usage is the following:

```
fprintf(pfile, "%12d%12d%14f", num1, num2, fnum1);
```

As you can see, the function name has an additional `f` (for *file*), compared with `printf()`, and the first argument is a file pointer that specifies the destination of the data to be written. The file pointer obviously needs to be set through a call to `fopen()` first. The remaining arguments are identical to that of `printf()`. This example writes the values of the three variables `num1`, `num2`, and `num3` to the file specified by the file pointer `pfile`, under control of the format string specified as the second argument. Therefore, the first two variables are of type `int` and are to be written with a field width of 12, and the third variable is of type `float` and is to be written to the file with a field width of 14.

Formatted Input from a File

You get formatted input from a file by using the function `fscanf()`. To read three variable values from a file `pfile` you would write this:

```
fscanf(pfile, "%12d%12d%14f", &num1, &num2, &fnum1);
```

This function works in exactly the same way as `scanf()` does with `stdin`, except that here you're obtaining input from a file specified by the first argument. The same rules govern the specification of the format string and the operation of the function as apply to `scanf()`. The function returns `EOF` if an error occurs such that no input is read; otherwise, it returns the number of values read as a value of type `int`.

TRY IT OUT: USING FORMATTED INPUT AND OUTPUT FUNCTIONS

You can demonstrate the formatted input and output functions with an example that will also show what's happening to the data in these operations:

```
/* Program 12.3 Messing about with formatted file I/O */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long num1 = 234567L;                    /* Input values...              */
  long num2 = 345123L;
  long num3 = 789234L;

  long num4 = 0L;                         /* Values read from the file... */
  long num5 = 0L;
  long num6 = 0L;

  float fnum = 0.0f;                      /* Value read from the file     */
  int   ival[6] = { 0 };                  /* Values read from the file    */
  FILE *pfile = NULL;                     /* File pointer                 */
  char *filename = "C:\\myfile.txt";
```

```
  pfile = fopen(filename, "w");                /* Create file to be written */
  if(pfile == NULL)
  {
    printf("Error opening %s for writing. Program terminated.", filename);
    exit(1);
  }
  fprintf(pfile, "%6ld%6ld%6ld", num1, num2, num3);   /* Write file          */
  fclose(pfile);                                      /* Close file          */
  printf("\n %6ld %6ld %6ld", num1, num2, num3);   /* Display values written */

  pfile = fopen(filename, "r");                       /* Open file to read   */
  fscanf(pfile, "%6ld%6ld%6ld", &num4, &num5 ,&num6); /* Read back           */
  printf("\n %6ld %6ld %6ld", num4, num5, num6);      /* Display what we got */

  rewind(pfile);                          /* Go to the beginning of the file */
  fscanf(pfile, "%2d%3d%3d%3d%2d%2d%3f", &ival[0], &ival[1], /* Read it again */
                    &ival[2], &ival[3], &ival[4] , &ival[5], &fnum);
  fclose(pfile);                                     /* Close the file and    */
  remove(filename);                                  /* delete physical file. */

  /* Output the results */
  printf("\n");
  for(int i = 0 ; i < 6 ; i++ )
    printf("%sival[i] = %d", i == 4 ? "\n\t" : "\t", i, ival[i]);
  printf("\nfnum = %f\n", fnum);
  return 0;
}
```

The output from this example is the following:

```
 234567 345123 789234
 234567 345123 789234
        ival[i] = 0     ival[i] = 1     ival[i] = 2     ival[i] =3
        ival[i] = 4     ival[i] = 5
fnum = 234.000000
```

How It Works

This example writes the values of `num1`, `num2`, and `num3`, which are defined and assigned values in their declaration, to the file `myfile.txt` on drive C. This is referenced through the pointer `pfile`. The file is closed and reopened for reading, and the values are read from the file in the same format as they are written. You then have the following statement:

```
rewind(pfile);
```

This statement calls the `rewind()` function, which simply moves the current position back to the beginning of the file so that you can read it again. You could have achieved the same thing by closing the file then reopening it again, but with `rewind()` you do it with one function call and the operation will be a lot faster.

Having repositioned the file, you read the file again with this statement:

```
fscanf(pfile, "%2d%3d%3d%3d%2d%2d%3f", &ival[0], &ival[1], /* Read it again */
                    &ival[2], &ival[3], &ival[4] , &ival[5], &fnum);
```

This statement reads the same data into the array `ival[]` and the variable `fnum`, but with different formats from those that you used for writing the file. You can see from the effects of this that the file consists of just a string of characters once it has been written, exactly the same as the output to the screen from `printf()`.

■ **Note** You can lose information if you choose a format specifier that outputs fewer digits precision than the stored value holds.

You can see that the values you get back from the file when you read it will depend on both the format string that you use and the variable list that you specify in the `fscanf()` function.

None of the intrinsic source information that existed when you wrote the file is necessarily maintained. Once the data is in the file, it's just a sequence of bytes in which the meaning is determined by how you interpret them. This is demonstrated quite clearly by this example, in which you've converted the original three values into eight new values.

Lastly, you leave everything neat and tidy in this program by closing the file and using the function `remove()` to delete it.

Dealing with Errors

The examples in this book have included minimal error checking and reporting because the code for comprehensive error checking and reporting tends to take up a lot of space in the book and make the programs look rather more complicated than they really are. In real-world programs, however, it's essential that you do as much error checking and reporting as you can.

Generally, you should write your error messages to `stderr`, which is automatically available to your program and always points to your display screen. Even though `stdout` may be redirected to a file by an operating system command, `stderr` continues to be assigned to the screen. It's important to check that a file you want to read does in fact exist and you have been doing this in the examples, but there's more that you can do. First of all, you can write error messages to `stderr` rather than `stdin`, for example

```
char *filename = "C:\\MYFILE.TXT";     /* File name    */
FILE *pfile = NULL;                    /* File pointer */

if(!(pfile = fopen(filename, "r")))
{
  fprintf(stderr, "\nCannot open %s to read it.", filename);
  exit(1);
}
```

The merit of writing to `stderr` is that the output will always be directed to the display and it will always be written immediately to the display device. This means that you will always see the output directed to `stderr`, regardless of what happens in the program. The `stdin` stream is buffered, so there is the risk that data could be left in the buffer and never displayed if your program crashes. Terminating a program by calling `exit()` ensures that output stream buffers will be flushed so output will be written to the ultimate destination. The stream `stdin` can be redirected to a file, but `stderr` can't be redirected simply to ensure that the output always occurs.

Knowing that some kind of error occurred is useful, but you can do more than this. The `perror()` function outputs a string that you pass as an argument plus an implementation-defined error message corresponding to the error that occurred. You could therefore rewrite the previous fragment as follows:

```
if(!(pfile = fopen(myfile, "r")))
{
  perror(strcat("Error opening ", filename));
  exit(1);
}
```

This will output your message consisting of the file name appended to the first argument to `strcat()`, plus a system-generated message relating to the error. The output will be written to `stderr`.

If an error occurs when you're reading a file, you can check whether the error is due to reaching the end of file. The `feof()` function will return a nonzero integer if the end of file has been reached, so you can check for this with statements such as these:

```
if(feof(pfile))
  printf("End of file reached.");
```

Note that I didn't write the message to `stderr` here because reaching the end of the file isn't necessarily an error.

The `ferror()` function returns a nonzero integer if an error occurrs with an operation on the stream that's identified by the file pointer that you pass as the argument. Calling this function enables you to establish positively that an error did occur. The `<errno.h>` header file defines a value with the name `errno` that may indicate what kind of file error has occurred. You need to read the documentation for your C implementation to find out the specifics of this. The value of `errno` may be set for errors other than just file operations.

You should always include some basic error checking and reporting code in all of your programs. Once you've written a few programs, you'll find that including some standard bits of code for each type of operation warranting error checks is no hardship. With a standard approach, you can copy most of what you need from one program to another.

Further Text File Operation Modes

Text mode is the default mode of operation with the open modes you have seen up to now, but in earlier versions of C you could specify explicitly that a file is to be opened in text mode. You could do this by adding `t` to the end of the existing specifiers. This gives you the mode specifiers `"wt"`, `"rt"`, and `"at"` in addition to the original three. I am only mentioning this because you may come across it in other C programs. Although most compilers will support this, it's not specifically part of the current C standard so it is best not to use this option in your code.

You can also open a text file for update—that is, for both reading and writing—using the specifier `"r+"`. You can also specify the open mode as `"w+"` if you want to both read and write a new file, or when you want to discard the original contents of an existing file before you start. Opening a file with the mode `"w+"` truncates the length of an existing file to zero, so only use this mode when you want to discard the current file contents. In older programs you may come across these modes written as `"rt+"` or `"r+t"` and `"wt+"` or `"w+t"`.

As I've said, in update mode you can both read and write a text file. However, you can't write to the file immediately after reading it or read from the file immediately after writing it, unless the `EOF` has been reached or the position in the file has been changed by some means. (This involves calling a function such as `rewind()` or some other function that modifies the file position.) The reason for this is that writing to a file doesn't necessarily write the data to the external device. It simply transfers

it to a buffer in memory that's written to the file once it's full, or when some other event causes it to be written. Similarly, the first read from a file will fill a buffer area in memory, and subsequent reads will transfer data from the buffer until it's empty, whereupon another file read to fill the buffer will be initiated. This is illustrated in Figure 12-3.

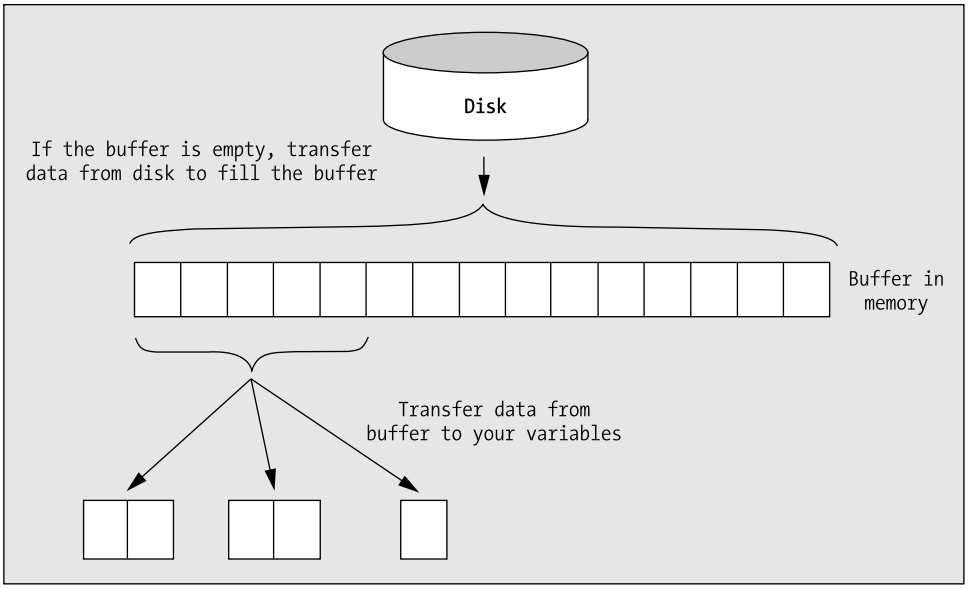


Figure 12-3. *Buffered input operations*

This means that if you were able to switch immediately from write mode to read mode, data would be lost because it would be left in the buffer. In the case of switching from read mode to write mode, the current position in the file may be different from what you imagine it to be, and you may inadvertently overwrite data on the file. A switch from read to write or vice versa, therefore, requires an intervening event that implicitly flushes the buffers. The `fflush()` function will cause the bytes remaining in an output buffer for the stream you pass as the argument to be written to an output file.

Binary File Input and Output

The alternative to text mode operations on a file is **binary mode**. In this mode, no transformation of the data takes place, and there's no need for a format string to control input or output, so it's much simpler than text mode. The binary data as it appears in memory is transferred directly to the file. Characters such as `'\n'` and `'\0'` that have specific significance in text mode are of no consequence in binary mode.

Binary mode has the advantage that no data is transformed or precision lost, as can happen with text mode due to the formatting process. It's also somewhat faster than text mode because no transformation operations are performed. The two modes are contrasted in Figure 12-4.

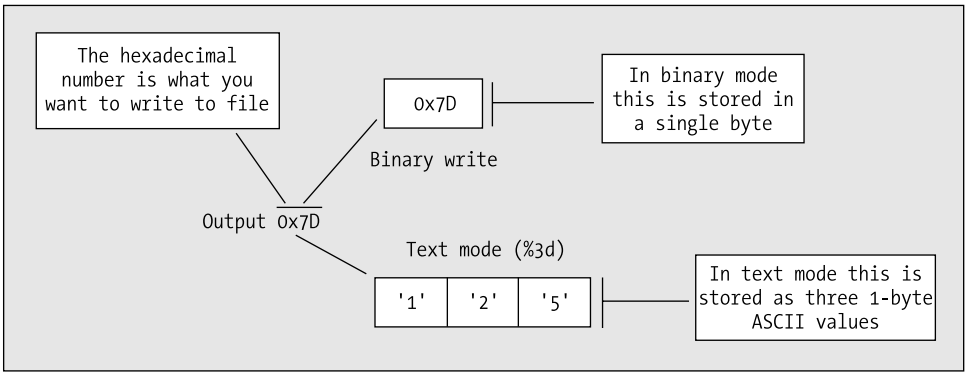


Figure 12-4. *Contrasting binary mode and text mode*

Specifying Binary Mode

You specify binary mode by appending `b` to the basic open mode specifiers I introduced initially. Therefore, you have the additional open mode specifiers `"wb"` for writing a binary file, `"rb"` to read a binary file, `"ab"` to append data to the end of a binary file, and `"rb+"` to enable reading and writing of a binary file.

Because binary mode involves handling the data to be transferred to and from the file in a different way from text mode, you have a new set of functions to perform input and output.

Writing a Binary File

You use the `fwrite()` function to write a binary file. This is best explained with an example of its use. Suppose that you open the file to be written with the following statements:

```
  char *filename = "myfile.bin";
FILE *pfile = fopen(filename, "wb");
```

The `filename` variable points to the string that defines the name of the file, and `pfile` is a variable to store a pointer to an object of type `FILE` as before.

You could write to the file with these statements:

```
long pdata[] = {2L, 3L, 4L};
int num_items = sizeof(pdata)/sizeof(long);
FILE *pfile = fopen(filename, "wb");
size_t wcount = fwrite(pdata, sizeof(long), num_items, pfile);
```

The `fwrite()` function operates on the principle of writing a specified number of binary data items to a file, where each item is a given number of bytes `long`. The first argument, `pdata`, is a pointer containing the starting address in memory of where the data items to be written are stored. The second argument specifies the size in bytes of each item to be written. The third argument, `num_items`, defines a count of the number of items to be written to the file. The file to which the data is to be transferred is identified by the last argument, `pfile`. The function `fwrite()` returns the count of the number of items actually written as a value of type `size_t`. If the operation is unsuccessful for some reason, this value will be less than `num_items`.

Note that there is no check that you opened the file in binary mode when you call the `fwrite()` function. The write operation will write binary data to a file that you open in text mode. Equally, there is nothing to prevent you from writing text data to a binary file. Of course, if you do this a considerable amount of confusion is likely to result.

The return value and the second and third arguments to the function are all of the same type as that returned by the `sizeof` operator. This is defined as type `size_t`, which you probably remember is an unsigned integer type.

The code fragment above uses the `sizeof` operator to specify the size in bytes of the objects to be transferred and also determines the number of items to be written using the expression `sizeof(pdata)/sizeof(long)`. This is a good way of specifying these values when this is possible, because it reduces the likelihood of error. Of course, in a real context, you should also check the return value in `wcount` to be sure the write is successful.

The `fwrite()` function is geared to writing a number of binary objects of a given length to a file. You can write in units of your own structures as easily as you can write values of type `int`, values of type `double`, or sequences of individual bytes.

This doesn't mean that the values you write in any given output operation all have to be of the same type. You might allocate some memory using `malloc()`, for instance, into which you assemble a sequence of data items of different types and lengths. You could then write the whole block of memory in one go as a sequence of bytes. Of course, when you come to read them back, you need to know the precise sequence and types for the values in the file if you are to make sense of them.

Reading a Binary File

You use the `fread()` function to read a binary file once it has been opened in read mode. Using the same variables as in the example of writing a binary file, you could read the file using a statement such as this:

```
size_t wcount = fread( pdata, sizeof(long), num_items, pfile);
```

This operates exactly as the inverse of the write operation. Starting at the address specified by `data`, the function reads `num_items` objects, each occupying the number of bytes specified by the second argument. The function returns the count of the number of items that were read. If the read isn't completely successful, the count will be less than the number of objects requested.

TRY IT OUT: READING A BINARY FILE

You can apply the binary file operations to a version of the Program 7.11 you saw in Chapter 7 for calculating primes. This time, you'll use a disk file as a buffer to calculate a larger number of primes. You can make the program automatically spill primes on to a disk file if the array assigned to store the primes is insufficient for the number of primes requested. In this version of the program to find primes, you'll improve the checking process a little.

In addition to the `main()`function that will contain the prime finding loop, you'll write a function to test whether a value is prime called `test_prime()`, a helper function that will check a given value against a block of primes called `check()`, and a function called `put_primes()`, which will retrieve the primes from the file and display them.

As this program consists of several functions and will work with variables at global scope, let's take a look at it piece by piece. We'll start with the function prototypes and global data before we look at detail of the functions:

```
/* Program 12.4  A prime example using binary files */
#include <stdio.h>
#include <stdlib.h>
#include <math.h >                    /* For square root function sqrt() */

/* Function prototypes */
int test_prime(unsigned long long N);
void put_primes(void);
int check(unsigned long long buffer[], size_t count, unsigned long long N);
```

```
/* Global data */
const unsigned int MEM_PRIMES = 100;  /* Count of number of primes in memory */

struct
{
  char *filename;                                  /* File name for primes   */
  FILE *pfile;                                     /* File stream pointer    */
  int nrec;                                        /* Number of file records */

  unsigned long long primes[MEM_PRIMES];           /* Array to store primes  */
  size_t index;                    /* Index of free location in array primes */

} global = { "C:\\myfile.bin",                     /* Physical file name     */
             NULL,                                 /* File pointer value     */
             0,                                    /* File record count      */
             {2ULL, 3ULL, 5ULL},                   /* primes array values    */
             3                                     /* Number of primes       */
           };

int main(void)
{
  /* Code for main()... */
}

/*********************************************************************
 * Function to test if a number, N, is prime using primes in       *
 * memory and on file                                              *
 * First parameter N - value to be tested                          *
 * Return value - a positive value for a prime, zero otherwise     *
 *********************************************************************/
int test_prime(unsigned long long N)
{
  /* Code for test_prime()... */
}

/*********************************************************************
 * Function to check whether an integer, N, is divisible by any    *
 * of the elements in the array pbuffer up to the square root of N.*
 * First parameter buffer - an array of primes                     *
 * second parameter count - number of elements in pbuffer          *
 * Third parameter N - the value to be checked                     *
 * Return value - 1 if N is prime, zero if N is not a prime,       *
 *               -1 for more checks                                *
 *********************************************************************/
int check(unsigned long long buffer[], size_t count, unsigned long long N)
{
  /* Code for check()... */
}
```

```
/*****************************************
 * Function to output primes from the file *
 *****************************************/
void put_primes(void)
{
  /* Code for put_primes()... */
}
```

After the usual `#include` statements, you have the prototypes for three functions used in the program:

```
int test_prime(unsigned long long N);
void put_primes(void);
int check(unsigned long long buffer[], size_t count, unsigned long long N);
```

You write just the parameter types within the prototypes without using a parameter name. Function prototypes can be written either with or without parameter names, but the parameter types must be specified. Generally, it's better to include names, because they give a clue to the purpose of the parameter. The names in the prototype can be different from the names used in the definition of the function, but you should only do this if it helps to make the code more readable. To allow the maximum range of possible prime values you'll store them as values of type `unsigned long long`.

It is often convenient to define variables at global scope when they need to be accessed by several functions. This avoids the need for long parameter lists for the functions where you would pass the data as arguments in each function call. However, the more names you declare at global scope, the greater the risk of collisions with local names in your program or names used in the standard libraries, so it is always a good idea to keep the number of names at global scope to a minimum. One way to reduce the number of names at global scope is to put your global variables inside a `struct`. Here we have defined a `struct` without a type name, and the `struct` variable has the name `global`. All the variables defined as members of the `struct` therefore do not appear at global scope because they must be qualified by the name `global`.

The first three members of the `struct` are `filename` that points to the name of the file that will store primes, the file stream pointer, `pfile`, and the `nrec` variable to store the number of records in the file. You then have the `primes` array that will hold up to `MEM_PRIMES` values in memory before they need to be written to the file followed by the `index` variable that records the current free element in the `primes` array.

You can see how we initialize the members of the `struct` using an initializer list. Note how the initial values for three elements in the `primes` array appear between braces; because there are only three initializers for the array, the remaining elements will be set to 0.

Here's the definition for `main()`:

```
int main(void)
{
  unsigned long long trial = 5ULL;                      /* Prime candidate */
  unsigned long num_primes = 3UL;                       /* Prime count     */
  unsigned long total = 0UL;                            /* Total required  */

  printf("How many primes would you like?  ");
  scanf("%lu", &total);                 /* Total is how many we need to find */
  total = total<4UL ? 4UL:total;        /* Make sure it is at least 4        */
```

```
  /* Prime finding and storing loop */
  while(num_primes < total)             /* Loop until we get total required  */
  {
    trial += 2ULL;                      /* Next value for checking           */
    if(test_prime(trial))               /* Check if trial is prime           */
    {                                   /* Positive value means prime        */
       global.primes[global.index++] = trial;  /* so store it                */
      num_primes++;                     /* Increment total number of primes  */

      if(global.index == MEM_PRIMES)    /* Check if array is full            */
      {
        /* File opened OK?   */
        if(!(global.pfile = fopen(global.filename, "ab")))
        { /* No, so explain and end the program */
          printf("\nUnable to open %s to append\n", global.filename);
          exit(1);
        }
        /* Write the array    */
        fwrite(global.primes, sizeof(unsigned long long),
                                            MEM_PRIMES, global.pfile);

        fclose(global.pfile);           /* Close the file                */
        global.index = 0U;              /* Reset count of primes in memory */
        global.nrec++;                  /* Increment file record count     */
      }
    }
  }

  if(total>MEM_PRIMES)                  /* If we wrote some to file         */
    put_primes();                       /* Display the contents of the file */
  if(global.index)                      /* Display any left in memory       */
    for(size_t i = 0; i<global.index ; i++)
    {
      if(i%5 == 0)
        printf("\n");                               /* Newline after five     */
      printf("%12llu", global.primes[i]);           /* Output a prime         */
    }

  if(total>MEM_PRIMES)                              /* Did we need a file?    */
    if(remove(global.filename))                     /* then delete it.        */
      printf("\nFailed to delete %s\n", global.filename); /* Delete failed   */
    else
      printf("\nFile %s deleted.\n", global.filename);    /* Delete OK       */
  return 0;
}
```

How It Works

The definition of the function `main()` follows the global declarations. When the program executes, you enter the number of primes you want to find, and this value controls the loop for testing prime candidates. Checking for a prime is performed by the function `test_prime()`, which is called in the `if` statement condition within the loop. The function returns 1 if the value tested is prime, and 0 otherwise. If a prime is found, then you execute these statements:

```
    global.primes[global.index++] = trial;   /* so store it               */
    num_primes++;                     /* Increment total number of primes  */
```

The first statement stores the prime that you've found in the `global.primes[]` array. You keep track of how many primes you have in total with the variable `num_primes`, and the `struct` member variable `global.index` records how many you have in memory at any given time.

Every time you find a prime and add it to the `primes[]` array, you perform the following check:

```
    if(global.index == MEM_PRIMES)    /* Check if array is full           */
    {
      /* File opened OK?   */
      if(!(global.pfile = fopen(global.filename, "ab")))
      { /* No, so explain and end the program */
        printf("\nUnable to open %s to append\n", global.filename);
        exit(1);
      }
      /* Write the array    */
      fwrite(global.primes, sizeof(unsigned long long),
                                          MEM_PRIMES, global.pfile);

      fclose(global.pfile);              /* Close the file                */
      global.index = OU;                 /* Reset count of primes in memory */
      global.nrec++;                     /* Increment file record count     */
    }
```

If you've filled the `global.primes` array, the `if` condition will be true and you'll execute the associated statement block. In this case, the file is opened in binary mode to append data. The first time this occurs, a new file will be created. On subsequent calls of the `fopen()` function file, the existing file will be opened with the current position set at the end of any existing data in the file, ready for the next block to be written. After writing a block, the file is closed, as it will be necessary to reopen it for reading in the function that performs the checking of prime candidates.

Finally in this group of statements, the count of the number of primes in memory is reset to 0 because they've all been safely stowed away, and the count of the number of blocks of primes written to the file is incremented.

When sufficient primes have been found to fulfill the number requested, you display the primes with the following statements:

```
  if(total>MEM_PRIMES)                 /* If we wrote some to file        */
    put_primes();                      /* Display the contents of the file */
  if(global.index)                     /* Display any left in memory      */
    for(size_t i = 0; i<global.index ; i++)
    {
      if(i%5 == 0)
        printf("\n");                            /* Newline after five    */
      printf("%12llu", global.primes[i]);        /* Output a prime        */
    }
```

It's quite possible that the number of primes requested can be accommodated in memory, in which case you won't write to a file at all. You must therefore check whether `total` exceeds `MEM_PRIMES` before calling the function `put_primes()` that outputs the primes in the file. If the value of `global.index` is positive, there are primes in the `global.primes` array that haven't been written to the file. In this case, you display these in the `for` loop, five to a line.

Finally in `main()`, you remove the file from the disk with the following statements:

```
  if(total>MEM_PRIMES)                          /* Did we need a file?    */
    if(remove(global.filename))                 /* then delete it.        */
      printf("\nFailed to delete %s\n", global.filename); /* Delete failed   */
    else
      printf("\nFile %s deleted.\n", global.filename);    /* Delete OK       */
```

The first `if` ensures that you don't attempt to delete the file if you didn't create one.

The implementation of the function to check whether a value is prime is as follows:

```
int test_prime(unsigned long long N)
{
  unsigned long long buffer[MEM_PRIMES]; /* local buffer for primes from file */

  int k = 0;

  if(global.nrec > 0)                           /* Have we written records? */
  {
    if(!(global.pfile = fopen(global.filename, "rb"))) /* Then open the file */
    {
      printf("\nUnable to open %s to read\n", global.filename);
      exit(1);
    }

    for(size_t i = 0; i < global.nrec ; i++)
    { /* Check against primes in the file first */
      /* Read primes */
      fread(buffer, sizeof( long long), MEM_PRIMES, global.pfile);
      if((k = check(buffer, MEM_PRIMES, N)) >= 0)  /* Prime or not?          */
      {
        fclose(global.pfile);                   /* Yes, so close the file */
        return k;                               /* 1 for prime, 0 for not */
      }
    }
    fclose(global.pfile);                       /* Close the file         */
  }

  /* Check against primes in memory */
  return check(global.primes, global.index, N);
}
```

The `test_prime()` function accepts a candidate value as an argument and returns 1 if it's prime and 0 if it isn't. If you've written anything to the file, this will be indicated by a positive value of `global.nrec`. In this case, the primes in the file need to be used as divisors first, because they are lower than those currently in memory as you compute them in sequence.

■ **Note** As you may remember, a prime is a number with no factors other than 1 and itself. It's sufficient to check whether a number is divisible by any of the primes less than the square root of the number to verify that it's prime. This follows from the simple logic that any exact divisor greater than the square root must have an associated factor (the result of the division) that's less than the square root.

To read the file, the function executes this statement:

```
fread(buffer, sizeof( long long), MEM_PRIMES, global.pfile);
```

This reads one block of primes from the file into the array `buffer`. The second argument defines the size of each object to be read, and `MEM_PRIMES` defines the number of objects of the specified size to be read.

Having read a block, the following check is executed:

```
if((k = check(buffer, MEM_PRIMES, N)) >= 0)  /* Prime or not?         */
{
  fclose(global.pfile);                      /* Yes, so close the file */
  return k;                                  /* 1 for prime, 0 for not */
}
```

Within the `if` condition, the `check()` function is called to determine if any of the array elements divide into the prime candidate with no remainder. This function returns 0 if an exact division is found, indicating the candidate isn't prime. If no exact division is found with primes up to the square root of the candidate value, 1 is returned, indicating that the candidate must be prime. Whatever value is returned from `check()`, in both cases you've finished checking so the file is closed and the same value is returned to `main()`.

The value –1 is returned from `check()` if no exact division has been found, but the square root of the test value hasn't been exceeded. You don't need to check for the –1 return explicitly, because it's the only possibility left if the value returned from `check()` isn't 0 or 1. In this case, the next block, if there is one, is read from the file in the next iteration of the `for` loop.

If the contents of the file have been exhausted without determining whether `N` is prime, the `for` loop will end, and you'll close the file and execute the following statement:

```
return check(global.primes, global.index, N);
```

Here, the test value is tried against any primes in the `primes` array in memory by the function `check()`. If a prime is found, the `check()` function will return 1; otherwise, the function will return 0. The value that's returned by the `check()` function will be returned to `main()`.

The code for the `check()` function is as follows:

```
int check(unsigned long long buffer[], size_t count, unsigned long long N)
{
  /* Upper limit */
  unsigned long long root_N = (unsigned long long)(1.0 + sqrt(N));

  for(size_t i = 0 ; i<count ; i++)
  {
    if(N % buffer[i] == 0ULL )              /* Exact division?             */
      return 0;                             /* Then not a prime           */

    if(buffer[i] > root_N)                  /* Divisor exceeds square root? */
      return 1;                             /* Then must be a prime       */
  }
  return -1;                                /* More checks necessary...   */
}
```

The role of this function is to check if any of the primes contained in the `buffer` array that's passed as the first argument divide exactly into the test value supplied as the second argument. The local variable in the function is declared in this statement:

```
  /* Upper limit */
  unsigned long long root_N = (unsigned long long)(1.0 + sqrt(N));
```

The integer variable, `root_N`, will hold the upper limit for divisors to be checked against the trial value. Only divisors less than the square root of the test value `N` are tried.

The checking is done in the `for` loop:

```
  for(size_t i = 0 ; i<count ; i++)
  {
    if(N % buffer[i] == 0UL )               /* Exact division?             */
      return 0;                             /* Then not a prime            */

    if(buffer[i] > root_N)                  /* Divisor exceeds square root? */
      return 1;                             /* Then must be a prime        */
  }
```

This loop steps through each of the divisors in the `buffer` array. If an exact divisor for `N` is found, the function will end and return 0 to indicate that the value isn't prime. If you arrive at a divisor that's greater than `root_N`, you've tried all those lower than this value, so `N` must be prime and the function returns 1. If the loop ends without executing a return statement then you haven't found an exact divisor but you haven't tried all values up to `root_N`. In this case the function returns –1 to indicate there's more checking to be done.

The last function that you need to define will output all the primes to the file:

```
void put_primes(void)
{
  unsigned long long buffer[MEM_PRIMES];      /* Buffer for a block of primes */

  if(!(global.pfile = fopen( global.filename, "rb"))) /* Open the file        */
  {
    printf("\nUnable to open %s to read primes for output\n", global.filename);
    exit(1);
  }

  for (size_t i = 0U ; i< global.nrec ; i++)
  {
    /* Read a block of primes   */
    fread(buffer, sizeof( unsigned long long), MEM_PRIMES, global.pfile);

    for(size_t j = 0 ; j<MEM_PRIMES ; j++)            /* Display the primes */
    {
      if(j%5 == 0U)                                   /* Five to a line     */
        printf("\n");
      printf("%12llu", buffer[j]);                    /* Output a prime     */
    }
  }
  fclose(global.pfile);                               /* Close the file     */
}
```

The operation of the `put_primes()` function is very simple. Once the file is opened, blocks of primes are read into the `buffer` array and as each is read, the `for` loop outputs the values to the screen, five to a line with a field width of 12. After all records have been read, the file is closed.

To run the program, you need to assemble all the functions that you've described into a single file and compile it. You'll be able to get as many primes as your computer and your patience permit.

A disadvantage of this program is that when you have a large number of primes, the output whizzes by on the screen before you can inspect it. You can do several things to fix this. You can write the output to the printer for a permanent record, instead of writing it to the screen. Or perhaps you can arrange for the program to display a prompt and wait for the user to press a key, between the output of one block and the next.

Moving Around in a File

For many applications, you need to be able to access data in a file other than in the sequential order you've used up to now. You can always find some information that's stored in the middle of a file by reading from the beginning and continuing in sequence until you get to what you want. But if you've written a few million items to the file, this may take some time.

Of course, to access data in random sequence requires that you have some means of knowing where the data that you would like to retrieve is stored in the file. Arranging for this is a complicated topic in general. There are many different ways of constructing pointers or indexes to make direct access to the data in a file faster and easier. The basic idea is similar to that of an index to a book. You have a table of keys that identify the contents of each record in the file you might want, and each key has an associated position in the file defined that records where the data is stored.

Let's look at the basic tools in the library that you need to enable you to deal with this kind of file input/output.

■ **Note** You cannot update a file in append mode. Regardless of any operations you may invoke to move the file position, all writes will be to the end of the existing data.

File Positioning Operations

There are two aspects to file positioning: finding out where you are at a given point in a file, and moving to a given point in a file. The former is basic to the latter: if you never know where you are, you can never decide how to get to where you want to go.

A random position in a file can be accessed regardless of whether the file concerned was opened in binary mode or in text mode. However, working with text mode files can get rather complicated in some environments, particularly Microsoft Windows. This is because the number of characters recorded in the file can be greater than the number of characters you actually write to the file. This is because a newline (`'\n'` character) in memory can translate into two characters when written to a file in text mode (carriage return, `CR`, followed by linefeed, `LF`). Of course, your C library function for reading the information sorts everything out when you read the data back. A problem only arises when you think that a point in the file is 100 bytes from the beginning. Whether writing 100 characters to a file in text mode results in 100 bytes actually appearing in the file depends on whether the data includes newline characters. If you subsequently want to write some different data that is the same length in memory as the original data written to the file, it will only be the same length on the file if it contains the same number of `'\n'` characters.

Thus writing to text files randomly is best avoided. For this reason, I'll sidestep the complications of moving about in text files and concentrate the examples on the much more useful—and easier—context of randomly accessing the data in binary files.

Finding Out Where You Are

You have two functions to tell you where you are in a file, both of which are very similar but not identical. They each complement a different positioning function. The first is the function `ftell()`, which has the prototype

```
long ftell(FILE *pfile);
```

This function accepts a file pointer as an argument and returns a `long` integer value that specifies the current position in the file. This could be used with the file that's referenced by the pointer `pfile` that you've used previously, as in the following statement:

```
fpos = ftell(pfile);
```

The `fops` variable of type `long` now holds the current position in the file and, as you'll see, you can use this in a function call to return to this position at any subsequent time. The value is actually the offset in bytes from the beginning of the file.

The second function providing information on the current file position is a little more complicated. The prototype of the function is the following:

```
int fgetpos(FILE *pfile, fpos_t *position);
```

The first parameter is your old friend the file pointer. The second parameter is a pointer to a type that's defined in `<stdio.h>` called `fpos_t`. `fpos_t` will be a type other than an array type that is able to record every position within a file. It is typically an integer type and with my library it is type `long`. If you're curious about what type `fpos_t` is on your system, then have a look at it in `<stdio.h>`.

The `fgetpos()` function is designed to be used with the positioning function `fsetpos()`, which I'll come to very shortly. The function `fgetpos()` stores the current position and file state information for the file in `position` and returns 0 if the operation is successful; otherwise, it returns a nonzero integer value. You could declare a variable `here` to be of type `fpos_t` with a statement such as this:

```
fpos_t here = 0;
```

You could now record the current position in the file with the statement

```
fgetpos(pfile, &here);
```

This records the current file position in the variable `here` that you have defined.

■ **Caution** Note that you must declare a variable of type `fpos_t`. It's no good just declaring a pointer of type `fpos_t*`, as there won't be any memory allocated to store the position data.

Setting a Position in a File

As a complement to `ftell()`, you have the function `fseek()`, which has the following prototype:

```
int fseek(FILE *pfile, long offset, int origin);
```

The first parameter is a pointer to the file that you're repositioning. The second and third parameters define where you want to go in the file. The second parameter is an offset from a reference point specified by the third parameter. The reference point can be one of three values that are specified by the predefined names `SEEK_SET`, which defines the beginning of the file; `SEEK_CUR`, which defines the current position in the file; and `SEEK_END`, which, as you might guess, defines the end of the file. Of course, all three values are defined in the header file `<stdio.h>`. For a text mode file, the

second argument must be a value returned by ftell() if you're to avoid getting lost. The third argument for text mode files must be SEEK_SET. So for text mode files, all operations with fseek() are performed with reference to the beginning of the file.

For binary files, the offset argument is simply a relative byte count. You can therefore supply positive or negative values for the offset when the reference point is specified as SEEK_CUR.

You have the fsetpos() function to go with fgetpos(). This has the rather straightforward prototype

```
int fsetpos(FILE *pfile, fpos_t *position);
```

The first parameter is a pointer to the file opened with fopen(), and the second is a pointer of the type you can see, where the value was obtained by calling fgetpos().

You can't go far wrong with this one really. You could use it with a statement such as this:

```
fsetpos(pfile, &here);
```

The variable here was previously set by a call to fgetpos(). As with fgetpos(), a nonzero value is returned on error. Because this function is designed to work with a value that is returned by fgetpos(), you can only use it to get to a place in a file that you've been before, whereas fseek() allows you to go to any specific position.

Note that the verb **seek** is used to refer to operations of moving the read/write heads of a disk drive directly to a specific position in the file. This is why the function fseek() is so named.

With a file that you've opened for update by specifying the mode as "rb+" or "wb+", for example, either a read or a write may be safely carried out on the file after executing either of the file positioning functions, fsetpos() or fseek(). This is regardless of what the previous operation on the file was.

TRY IT OUT: RANDOM ACCESS TO A FILE

To exercise your newfound skills with files, you can create a revised version of Program 11.6 from the previous chapter to allow you to keep a dossier on family members. In this case, you'll create a file containing data on all family members, and then you'll process the file to output data on each member and that member's parents. The structures used only extend to a minimum range of members in each case. You can, of course, embellish these to hold any kind of scuttlebutt you like on your relatives.

Let's look at the function main() first:

```
/* Program 12.5 Investigating the family.*/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Global Data */
const int NAME_MAX = 20;

struct
{
  char *filename;                          /* Physical file name      */
  FILE *pfile;                             /* File pointer            */
} global = {"C:\\myfile.bin", NULL };
```

```
/* Structure types */
struct Date                                  /* Structure for a date        */
{
  int day;
  int month;
  int year;
};

typedef struct family                        /* Structure for family member */
{
  struct Date dob;
  char name[NAME_MAX];
  char pa_name[NAME_MAX];
  char ma_name[NAME_MAX];
}Family;

/* Function prototypes */
bool get_person(Family *pfamily);               /* Input function           */
void getname(char *name);                       /* Read a name              */
void show_person_data(void);                    /* Output function          */
void get_parent_dob(Family *pfamily);           /* Function to find pa & ma */

int main(void)
{
  Family member;                              /* Stores a family structure */

  if(!(global.pfile = fopen(global.filename, "wb")))
  {
    printf("\nUnable to open %s for writing.\n", global.filename);
    exit(1);
  }

  while(get_person(&member))                    /* As long as we have input */
    fwrite(&member, sizeof member, 1, pfile);   /*    write it away         */

  fclose(global.pfile);                   /* Close the file now its written */

  show_person_data();                     /* Show what we can find out      */

  if(remove(global.filename))
    printf("\nUnable to delete %s.\n", global.filename);
  else
    printf("\nDeleted %s OK.\n", global.filename);
  return 0;
}
```

How It Works

After the `#include` statements, you have the global variables and structure definitions. You've seen all of these in previous examples. `Family` isn't a variable, but it has been declared as a type name for the structure `family`. This will allow you to declare `Family` type objects without having to use the keyword `struct`. Following the structure declarations, you have the prototypes for the three functions that you're using in addition to `main()`.

Because you want to try out the file-positioning functions in addition to the basic file read and write operations, the example has been designed to exercise these as well. You get the input on one person at a time in the `get_person()` function, where the data is stored in the member structure object. You write each structure to the file as soon as it is received, and the input process ceases when the function `get_person()` returns 0.

When the `while` loop ends, the input file is closed and the function `show_person_data()` is called. You use the file position getting and setting functions within this function. Lastly, the file is deleted from the disk by the function `remove()`.

The code for the input function, `get_person()`, is as follows:

```
/* Function to input data on Family members */
bool get_person(Family *temp)
{
  static char more = '\0';                   /* Test value for ending input */

  printf("\nDo you want to enter details of a%s person (Y or N)? ",
                                      more != '\0'?"nother " : "" );
  scanf(" %c", &more);

  if(tolower(more) == 'n')
    return false;

  printf("\nEnter the name of the person: ");
  getname(temp->name);                           /* Get the person's name */
  printf("\nEnter %s's date of birth (day month year); ", temp->name);
  scanf("%d %d %d", &temp->dob.day, &temp->dob.month, &temp->dob.year);

  printf("\nWho is %s's father? ", temp->name);
  getname(temp->pa_name);                        /* Get the father's name  */
  printf("\nWho is %s's mother? ", temp->name);
  getname(temp->ma_name);                        /* Get the mother's name  */
return true;
}
```

This function is fairly self-explanatory. None of the mechanisms involved in this function is new to you. An indicator, `more`, controls whether reading data continues, and it's set by the input following the first prompt. It's defined as `static` so the variable and its value persists in the program from one call of `get_person()` to the next. This allows the prompt to work correctly in selecting a slightly different message for the second and subsequent iterations.

If no data input takes place, which is triggered when **N** or **n** is entered in response to the initial prompt in the function, `false` is returned. If more data entry occurs, it's entered into the appropriate structure members and `true` is returned.

The names are read by the `getname()` function that can be implemented like this:

```
/* Read a name from the keyboard */
void getname(char *name)
{
  fflush(stdin);                             /* Skip whitespace        */
  fgets(name, NAME_MAX, stdin);
  int len = strlen(name);
  if(name[len-1] == '\n')                    /* If last char is newline */
    name[len-1] = '\0';                      /* overwrite it           */
}
```

This reads a name using `fgets()` to allow whitespace in the input and also to ensure the capacity of the storage for a name is not exceeded. If the input exceeds `NAME_MAX` characters, including the terminating null, the name will be truncated. The `fgets()` function stores the newline that arises when the Enter key is pressed, so when this is present as the last string character, you overwrite it with a terminating null.

The next function generates the output for each person, including the date of birth of both parents, if they've been recorded. The code for this function is as follows:

```
/* Function to output data on people on file   */
void show_person_data(void)
{
  Family member;                      /* Structure to hold data from file  */
  fpos_t current = 0;                 /* File position                     */

  /* Open file for binary read */
  if(!(global.pfile = fopen(global.filename, "rb")))
  {
    printf("\nUnable to open %s for reading.\n", global.filename);
    exit(1);
  }

  /* Read data on person */
  while(fread(&member, sizeof member, 1, global.pfile))
  {
    fgetpos(global.pfile, &current);   /* Save current position            */
    printf("\n\n%s's father is %s, and mother is %s.",
             member.name, member.pa_name, member.ma_name);
    get_parent_dob(&member);           /* Get parent data                  */
    fsetpos(global.pfile, &current);   /* Position file to read next       */
  }
   fclose(global.pfile);               /* Close the file                   */
}
```

This function processes each structure in sequence from the file.

After declaring a variable of type `Family`, you declare a variable, `current`, with the following statement:

```
  fpos_t current = 0;                  /* File position                     */
```

This statement declares `current` as type `fpos_t`. This variable will be used to remember the current position in the file. The `get_parent_dob()` function is called later in this function, which also accesses the file. It's therefore necessary to remember the file position of the next structure to be read on each iteration before calling `get_parent_dob()`.

After opening the file for binary read operations, all of the processing takes place in a loop controlled by the following:

```
  while(fread(&member, sizeof member, 1, global.pfile))
```

This uses the technique of reading the file within the loop condition and using the value returned by the function `fread()` as the determinant of whether the loop continues. If the function returns 1, the loop continues, and when 0 is returned, the loop is terminated.

Within the loop you have these statements:

```
    fgetpos(global.pfile, &current);   /* Save current position            */
    printf("\n\n%s's father is %s, and mother is %s.",
             member.name, member.pa_name, member.ma_name);
    get_parent_dob(&member);           /* Get parent data                  */
    fsetpos(global.pfile, &current);   /* Position file to read next       */
```

First, the current position in the file is saved, and the parents of the current person stored in `member` are displayed. You then call the function `get_parent_dob()`, which will search the file for parent entries. On returning after the call to this function, the file position is unknown, so a call to `fsetpos()` is made to restore it to the position required for the next structure to be read. After all the structures have been processed, the `while` loop terminates and the file is closed.

The function to find the dates of birth for the parents of an individual is as follows:

```
/* Function to find parents' dates of birth. */
void get_parent_dob(Family *pmember)
{
  Family relative;                          /* Stores a relative         */
  int num_found = 0;                        /* Count of relatives found  */

  rewind(global.pfile);                     /* Set file to the beginning */

  /* Get the stuff on a relative */
  while(fread(&relative, sizeof(Family), 1, global.pfile))
  {
    if(strcmp(pmember->pa_name, relative.name) == 0)   /*Is it pa?        */
    { /* We have found dear old dad */
      printf("\n Pa was born on %d/%d/%d.",
            relative.dob.day, relative.dob.month, relative.dob.year);

      if(++num_found == 2)                  /* Increment parent count    */
        return;                             /* We got both so go home    */
    }
    else
      if(strcmp(pmember->ma_name, relative.name) == 0) /*Is it ma?        */
      { /* We have found dear old ma */
        printf("\n Ma was born on %d/%d/%d.",
               relative.dob.day, relative.dob.month, relative.dob.year);

        if(++num_found == 2)                /* Increment parent count    */
           return;                          /* We got both so go home    */
      }
  }
}
```

As the file has already been opened by the calling program, it's only necessary to set it back to the beginning with the `rewind()` function before beginning processing. The file is then read sequentially, searching each structure that's read for a match with either parent name. The search mechanism for the father is contained in the following statements:

```
    if(strcmp(pmember->pa_name, relative.name) == 0)   /*Is it pa?        */
    { /* We have found dear old dad */
      printf("\n Pa was born on %d/%d/%d.",
            relative.dob.day, relative.dob.month, relative.dob.year);
```

```
      if(++num_found == 2)                    /* Increment parent count    */
        return;                               /* We got both so go home    */
    }
```

The name entry for the father of the person indicated by `pmember` is compared with the name member in the structure object `relative`. If the father check fails, the function continues with an identical mother check.

If a parent is found, the date of birth information is displayed. A count is kept in `num_found` of the number of parents discovered in the file, and the function is exited if both have been found. The function ends in any event after all structures have been read from the file.

To run this program, you need to assemble the code for `main()` and the other functions into a single file. You can then compile and execute it. Of course, the example could be written equally well using `ftell()` and `fseek()` as positioning functions.

As in the previous examples in this chapter, the program uses a specific file name, on the assumption that the file doesn't already exist when the program is run. There's a way in C to create temporary files that get around this so let's look into that next.

Using Temporary Work Files

Very often you need a work file just for the duration of a program. You use it only to store intermediate results and you can throw it away when the program is finished. The program that calculates primes in this chapter is a good example; you really only need the file during the calculation.

You have a choice of two functions to help with temporary file usage, and each has advantages and disadvantages.

Creating a Temporary Work File

The first function will create a temporary file automatically. Its prototype is the following:

```
FILE *tmpfile(void);
```

The function takes no arguments and returns a pointer to the temporary file. If the file can't be created for any reason—for example, if the disk is full—the function returns `NULL`. The file is created and opened for update, so it can be written and read, but obviously it needs to be in that order. You can only ever get out what you have put in. The file is automatically deleted on exit from your program, so there's no need to worry about any mess left behind. You'll never know what the file is called, and because it doesn't last this doesn't matter.

The disadvantage of this function is that the file will be deleted as soon as you close it. This means you can't close the file, having written it in one part of the program, and then reopen it in another part of the program to read the data. You must keep the file open for as long as you need access to the data. A simple illustration of creating a temporary file is provided by these statements:

```
FILE pfile;                           /* File pointer                 */
pfile = tmpfile();                    /* Get pointer to temporary file */
```

Creating a Unique File Name

The second possibility is to use a function that provides you with a unique file name. Whether this ends up as the name of a temporary file is up to you. The prototype for this function is the following:

```
char *tmpnam(char *filename);
```

If the argument to the function is NULL, the file name is generated in an internal static object, and a pointer to that object is returned. If you want the name stored in a char array that you declare yourself, it must be at least L_tmpnam characters long, where L_tmpnam is an integer constant that is defined in <stdio.h>. In this case, the file name is stored in the array that you specify as an argument, and a pointer to your array is also returned. If the function is unable to create a unique name, it will return NULL.

So to take the first possibility, you can create a unique file with the following statements:

```
FILE *pFile = NULL;
char *filename = tmpnam(NULL);
if(filename != NULL)
  pfile = fopen(filename, "wb+");
```

Here you declare your file pointer pfile and then your pointer filename that is initialized with the address of the temporary file name that the tmpnam() function returns. Because the argument to tmpnam() is NULL, the file name will be generated as an internal static object whose address will be placed in the pointer filename. As long as filename is not NULL you call fopen() to create the file with the mode "wb+". Of course, you can also create temporary text files, too.

Don't be tempted to write this:

```
pfile = fopen(tmpnam(NULL), "wb+");    /* Wrong!! */
```

Apart from the fact there is a possibility that tmpnam() may return NULL, you also no longer have access to the file name, so you can't use remove() to delete the file.

If you want to create the array to hold the file name yourself, you could write this:

```
FILE *pfile = NULL;
char filename[L_tmpnam];
if(tmpnam(filename) != NULL)
  pfile = fopen(filename, "wb+");
```

Remember, the assistance that you've obtained from the standard library is just to provide a unique name. It's your responsibility to delete any files created.

■ **Note** You should note that you'll be limited to a maximum number of unique names from this function in your program. You can access the maximum number through TMP_MAX that is defined in <stdio.h>.

Updating Binary Files

You have three open modes that provide for updating binary files:

- The mode "r+b" (or you can write it as "rb+") opens an existing binary file for both reading and writing. With this open mode you can read or write anywhere in the file.
- The mode "w+b" (or you can write "wb+") truncates the length of an existing binary file to zero so the contents will be lost; you can then carry out both read and write operations but, obviously because the file length is zero, you must write something before you can read the file. If the file does not exist, a new file will be created when you call fopen() with mode "w+b".
- The third mode "a+b" (or "ab+") opens an existing file for update. This mode only allows write operations at the end of the file.

While you can write each of the open modes for updating binary files in either of two ways, I prefer to always put the + at the end because for me it is more obvious that the + is significant and means update. We can first put together an example that uses mode "wb+" to create a new file that we can then update using the other modes.

TRY IT OUT: WRITING A BINARY FILE WITH AN UPDATE MODE

The file will contain names of people and their ages, the data being read from the keyboard. A name will be stored as a single string containing a first name and a second name. I have specified a full file path to the temp directory on drive C: in the code so you should check that this will be satisfactory on your system or change it to suit your environment. Note that if the directory in the path does not exist, the program will fail. Here's the code for the example:

```
/* Program 12.6 Writing a binary file with an update mode */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

const int MAXLEN = 30;                    /* Size of name buffer    */

void listfile(char *filename);            /* List the file contents */

int main(void)
{
  const char *filename = "C:\\temp\\mydata.bin";
  char name[MAXLEN];                          /* Stores a name    */
  size_t length = 0;                          /* Length of a name */
  int age = 0;                                /* Person's age     */
  char answer = 'y';

  FILE *pFile = fopen(filename, "wb+");

  do
  {
    fflush(stdin);                            /* Remove whitespace */

    printf("\nEnter a name less than %d characters:", MAXLEN);
    gets(name);                               /* Read the name     */

    printf("Enter the age of %s: ", name);
    scanf(" %d", &age);                       /* Read the age      */

    /* Write the name & age to file */
    length = strlen(name);                    /* Get name length   */
    fwrite(&length, sizeof(length), 1, pFile);  /* Write name length */
    fwrite(name, sizeof(char), length, pFile);  /* then the name     */
    fwrite(&age, sizeof(age), 1, pFile);        /* then the age      */

    printf("Do you want to enter another(y or n)?  " );
    scanf("\n%c", &answer);
  } while(tolower(answer) == 'y');
```

```
  fclose(pFile);                              /* Close the file    */

  listfile(filename);                         /* List the contents */
  return 0;
}

/* List the contents of the binary file */
void listfile(char *filename)
{
  size_t length = 0;                          /* Name length       */
  char name[MAXLEN];                          /* Stores a name     */
  int age = 0;
  char format[20];                            /* Format string     */

  /* Create the format string for names up to MAXLEN long */
  sprintf(format, "\n%%-%ds Age:%%4d", MAXLEN);

  FILE *pFile = fopen(filename, "rb");        /* Open to read      */
  printf("\nThe contents of %s are:", filename);

  /* Read records as long as we read a length value */
  while(fread(&length, sizeof(length), 1, pFile) == 1)
  {
    if(length+1>MAXLEN)
    {
      printf("\nName too long.");
      exit(1);
    }
    fread(name, sizeof(char), length, pFile);   /* Read the name     */
    name[length] = '\0';                        /* Append terminator */
    fread(&age, sizeof(age), 1, pFile);         /* Read the age      */
    printf(format, name, age);                  /* Output the record */
  }
  fclose(pFile);
}
```

Here's some sample output from this program:

```
Enter a name less than 30 characters:Bill Bloggs
Enter the age of Bill Bloggs: 21
Do you want to enter another(y or n)?  y

Enter a name less than 30 characters:Yolande Dogsbreath
Enter the age of Yolande Dogsbreath: 27
Do you want to enter another(y or n)?  y

Enter a name less than 30 characters:Ned Nudd
Enter the age of Ned Nudd: 33
Do you want to enter another(y or n)?  y

Enter a name less than 30 characters:Binkie Huckerback
Enter the age of Binkie Huckerback: 18
Do you want to enter another(y or n)?  y
```

```
Enter a name less than 30 characters:Mary Dunklebiscuit
Enter the age of Mary Dunklebiscuit: 29
Do you want to enter another(y or n)?  n

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age:  21
Yolande Dogsbreath             Age:  27
Ned Nudd                       Age:  33
Binkie Huckerback              Age:  18
Mary Dunklebiscuit             Age:  29
```

How It Works

The file is opened for binary update operations with the mode specified as `"rb+"`. In this mode the file contents will be overwritten because the file length is truncated to zero. If the file does not exist, a file will be created. The data is read from the keyboard and the file is written in the `do-while` loop. The first statement in the loop flushes `stdin`:

```
fflush(stdin);                                /* Remove whitespace */
```

This is necessary because the read operation for a single character that appears in the loop condition will leave a newline character in `stdin` on all loop iterations after the first. If you don't get rid of this character, the read operation for the name will not work correctly because the newline will be read as an empty name string.

After reading a name and an age from the keyboard, the information is written to the file as binary data with these statements:

```
length = strlen(name);                      /* Get name length   */
fwrite(&length, sizeof(length), 1, pFile);  /* Write name length */
fwrite(name, sizeof(char), length, pFile);  /* then the name     */
fwrite(&age, sizeof(age), 1, pFile);        /* then the age      */
```

The names are going to vary in length and you have basically two ways to deal with this. You can write the entire name array to the file each time and not worry about the length of a name string. This is simpler to code but means that there would be a lot of spurious data in the file. The alternative is to adopt the approach used in the example. The length of each name string is written preceding the name itself, so to read the file you will first read the length, then read that number of characters from the file as the name. Note that the `'\0'` string terminator is not written to the file, so you have to remember to add this at the end of each name string when you read the file back.

The loop allows as many records as you want to be added to the file because it continues as long as you enter `'y'` or `'Y'` when prompted. When the loop ends you close the file and call the `listfile()` function that lists the contents of the file on `stdout`.

The `listfile()` function opens the file for binary read operations with the mode `"rb"`. In this mode the file pointer will be positioned at the beginning of the file and you can only read it.

The maximum length of a name is specified by the `MAXLEN` symbol so it would be helpful to use the format `%-MAXLENs` for outputting names. This would output a name left-justified in a field that has a width that is the maximum name length, so the names would line up nicely and they would always fit in the field. Of course, you can't really write this as part of the format string because the letters in the `MAXLEN` symbol name would be interpreted as just that, a sequence of letters, and not the value of the symbol. To achieve the required result, the `listfile()` function uses the `sprintf()` function to write to the `format` array to create a format string:

```
sprintf(format, "\n%%-%ds Age:%%4d", MAXLEN);
```

The `sprintf()` function works just like `printf()` except that the output is written to an array of `char` elements that you specify as the first argument. This operation therefore writes the value of `MAXLEN` to the `format` array, using the format string:

```
"\n%%-%ds Age:%%4d"
```

After the \n for a newline there is %% which specifies a single % symbol in the output. The - will appear next in the output followed by the value of MAXLEN formatted using the %d specification. This will be followed by s, then a space followed by Age:. Finally the output will contain a % character followed by 4d. Because the MAXLEN symbol is defined as 30, after executing the sprintf() function the format array will contain the following string:

```
"\n%-30s Age:%d"
```

The file is read and the contents listed on stdout in the while loop that is controlled by the value of an expression that reads the name length from the file:

```
while(fread(&length, sizeof(length), 1, pFile) == 1)
{
...
}
```

The call to fread() reads one item of sizeof(length) bytes into the location specified by &length. When the operation is successful the fread() function returns the number of items read but when the end-of-file is reached the function will return less than the number requested because there is not more data to be read. Thus when we reach the end of file, the loop will end.

An alternative way of recognizing when the end-of-file is reached is to code the loop like this:

```
while(true)
{
  fread(&length, sizeof(length), 1, pFile);
  /* Now check for end of file */
  if(feof(pFile))
    break;
...
}
```

The feof() function tests the end-of-file indicator for the stream specified by the argument and returns nonzero if the indicator is set. Thus when end-of-file is reached, the break will be executed and the loop will end.

After reading the length value from the file, you check that you have space to accommodate the name that follows with the following statements:

```
if(length+1>MAXLEN)
{
  printf("\nName too long.");
  exit(1);
}
```

Remember that the name in the file does not have a terminating '\0' character so you have to allow for that in the name array. Hence you compare length+1 with MAXLEN.

You read the name and age from the file with these statements:

```
fread(name, sizeof(char), length, pFile);   /* Read the name     */
name[length] = '\0';                        /* Append terminator */
fread(&age, sizeof(age), 1, pFile);         /* Read the age      */
```

Finally in the loop, you write the name and age to stdout using the format string that you created using the sprintf() function.

```
printf(format, name, age);                  /* Output the record */
```

Changing the File Contents

We could revise and extend the previous example so that it uses the other two binary update modes. Let's add capability to update the existing records in the file as well as add records or delete the file. This program will be rather more complicated so it will be helpful to break the operations down into more functions. We will still write the file so the names are recorded as they are, so the records consisting of a name and an age will vary in length. This will provide an opportunity to see some of the complications this introduces when we want to change the contents of the file.

To give you an idea of where we are headed, let's look at the program in outline. The program will consist of the following nine functions:

`main()`: Controls overall operation of the program and allows the user to select from a range of operations on the file.

`listfile()`: Outputs the contents of the file to `stdin`.

`writefile()`: Operates in two modes, either writes a new file with records read from `stdin`, or appends a record to the existing file.

`getrecord()`: Reads a record from `stdin`.

`getname()`: Reads a name from `stdin`.

`writerecord()`: Writes a record to the file.

`readrecord()`: Reads a record from the file.

`findrecord()`: Find the record in the file with a name that matches input.

`duplicatefile()`: Reproduces the file replacing a single updated record. This function is used to update a record when the new record will be a different length from the record being replaced.

Figure 12-5 shows the call hierarchy for the functions in the application.

The three functions called by `main()` implement the basic functionality of the program. The functions to the right of these three provide functionality that helps to simplify the three primary functions.

It will simplify the code if we define a structure that we can use to pass a name and age between functions:

```
struct Record
{
  char name[MAXLEN];
  int age;
};
```

We could easily write objects of type `Record` to the file, but this would mean the whole `name` array of `MAXLEN` elements would be written each time, so the file would contain a lot of spurious bytes for names shorter than `MAXLEN` characters. However, the structure will provide a very convenient way of passing a name and the associated age value to a function. There will be several functions in the new example, so let's first look at the code for each function before we put the example together. You can assemble the code for the functions into a single source file as you read through the following sections.

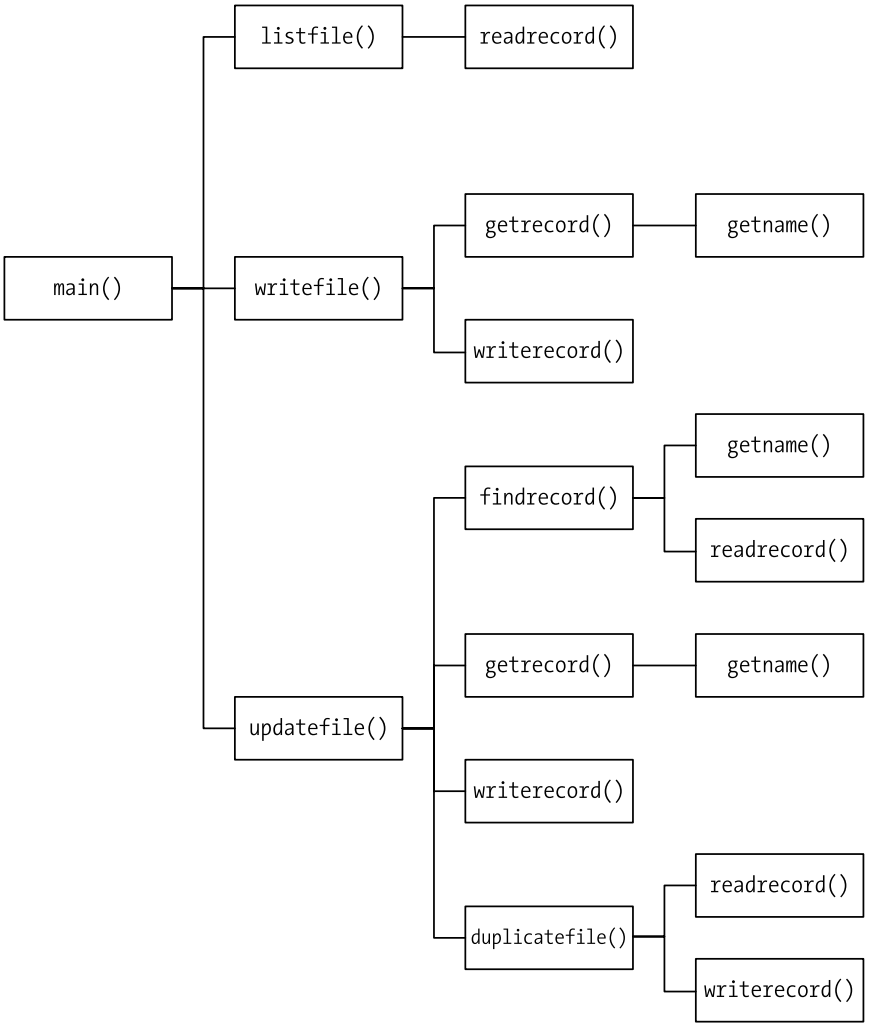


Figure 12-5. *The hierarchy of function calls in Program 12.8*

Reading a Record from the Keyboard

We can write a function that will read a name string and age value from `stdin` and store them in a `Record` object. The prototype of the function will be the following:

```
struct Record *getrecord(struct Record *precord);
```

The function requires an argument that is a pointer to an existing `Record` structure object and it returns the address of the same object. By returning the address of the `Record` object you make it possible to use a call to this function as an argument to another function that expects an argument of type `Record *`.

Here's how the implementation of the function looks:

```
/* Read the name and age for a record from the keyboard */
struct Record *getrecord(struct Record *precord)
{
  /* Verify the argument is good */
  if(!precord)
  {
    printf("No Record object to store input.");
    return NULL;
  }

  printf("\nEnter a name less than %d characters:", MAXLEN);
  getname(precord->name);                    /* readf the name    */

  printf("Enter the age of %s: ", precord->name);
  scanf(" %d", &precord->age);               /* Read the age      */
  return precord;
}
```

This is a straightforward operation where the name and age that are read from `stdin` are stored in the appropriate members of the `Record` object that is pointed to by `precord`. The name is read by the auxiliary function `getname()` that you can implement like this:

```
/* Read a name from the keyboard */
void getname(char *pname)
{
  fflush(stdin);
  fgets(pname, MAXLEN, stdin);        /* Read the name     */
  int len = strlen(pname);
  if(pname[len-1] == '\n')         /* if there's a newline */
    pname[len-1] = '\0';           /* overwrite it         */
}
```

The only slight complication in `getname()` is the need to deal with the `'\n'` that may be stored by the `fgets()` function. If the input exceeds `MAXLEN` characters then the `'\n'` will still be in the input buffer and not stored in the array pointed to by `pname`. You'll need to read a name at more that one location in the program so packaging the operation in the `getname()` function is convenient.

Writing a Record to a File

You can now define a function that will write the members of a record object to a file identified by a pointer of type `FILE *`. The prototype would look like this:

```
void writerecord(struct Record *precord, FILE *pFile);
```

The first parameter is a pointer to a `Record` structure that has the name and age that are to be written to the file as members. The second argument is the file pointer.

The implementation looks like this:

```
/* Write a new record to the file at the current position */
void writerecord(struct Record *precord, FILE *pFile)
{
```

```
  /* Verify the arguments are good */
  if(!precord)
  {
    printf("No Record object to write to the file.");
    return;
  }
  if(!pFile)
  {
    printf("No stream pointer for the output file.");
    return;
  }

  /* Write the name & age to file */
  size_t length = strlen(precord->name);                 /* Get name length   */
  fwrite(&length, sizeof(length), 1, pFile);             /* Write name length */
  fwrite(precord->name, sizeof(char), length, pFile);    /* then the name     */
  fwrite(&precord->age, sizeof(precord->age), 1, pFile); /* then the age      */
}
```

The function checks that the file pointer exists and the file will be written at the current position. It is therefore the responsibility of the calling function to ensure that the file has been opened in the correct mode and the file position has been set appropriately. The function first writes the length of the string to the file, followed by the string itself, excluding the terminating `'\0'`. This is to enable the code that will read the file to determine first how many characters are in the name string. Finally the age value is written to the file.

Reading a Record from a File

Here's the prototype of a function to read a single record from a file:

```
struct Record *readrecord(struct Record *precord, FILE *pFile);
```

The file to be read is identified by the second parameter, a file pointer. Purely as a convenience, the return value is the address that is passed as the first argument.

The implementation of the `readrecord()` function looks like this:

```
/* Reads a record from the file at the current position */
struct Record * readrecord(struct Record *precord, FILE *pFile)
{
  /* Verify the arguments are good */
  if(!precord)
  {
    printf("No Record object to store data from the file.");
    return NULL;
  }
  if(!pFile)
  {
    printf("No stream pointer for the input file.");
    return NULL;
  }

  size_t length = 0;                                  /* Name length      */
  fread(&length, sizeof(length), 1, pFile);           /* Read the length  */
  if(feof(pFile))                                     /* If it's end file */
    return NULL;                                      /* return NULL      */
```

```
  /* Verify the name can be accommodated */
  if(length+1>MAXLEN)
  {
    fprintf(stderr, "\nName too long. Ending program.");
    exit(1);
  }

  fread(precord->name, sizeof(char), length, pFile);    /* Read the name     */
  precord->name[length] = '\0';                         /* Append terminator */
  fread(&precord->age, sizeof(precord->age), 1, pFile); /* Read the age      */

  return precord;
}
```

Like the `writerecord()` function, the `readrecord()` function assumes the file has been opened with the correct mode specified and by default attempts to read a record from the current position. Each record starts with a length value that is read first. Of course, the file position could be at the end of the file, so you check for `EOF` by calling `feof()` with the file pointer as the argument after the read operation. If it is the end-of-file, the `feof()` function returns a nonzero integer value, so in this case you return `NULL` to signal the calling program that `EOF` has been reached.

The function then checks for the possibility that the length of the name exceeds the length of the name array. If it does, the program ends after outputting a message to the standard error stream.

If all is well, the name and age are read from the file and stored in the members of the record object. A `'\0'` has to be appended to the `name` string to avoid disastrous consequences when working with the string subsequently.

Writing a File

Here's the prototype of a function that will write an arbitrary number of records to a file, where the records are entered from the keyboard:

```
void writefile(char *filename, char *mode);
```

The first parameter is the name of the file to be written, so this implies that the function will take care of opening the file. The second parameter is the file open mode to be used. By passing `"wb+"` as the mode, the `writefile()` function will write to a file discarding any existing contents or create a new file with the specified name if it does not already exist. If the mode is specified as `"ab+"`, records will be appended to an existing file, and a new file will be created if there isn't one already.

Here's the implementation of the function:

```
/* Write to a file */
void writefile(char *filename, char *mode)
{
  char answer = 'y';

  FILE *pFile = fopen(filename, mode);    /* Open the file                */
  if(pFile == NULL)                       /* Verify file is open          */
  {
    fprintf(stderr, "\n File open failed.");
    exit(1);
  }
```

```
  do
  {
    struct Record record;                /* Stores a record name & age    */

    writerecord(getrecord(&record), pFile); /* Get record & write the file */

    printf("Do you want to enter another(y or n)?  " );
    scanf("\n%c", &answer);
    fflush(stdin);                       /* Remove whitespace             */
  } while(tolower(answer) == 'y');

  fclose(pFile);                         /* Close the file                */
}
```

After opening the file with the mode passed as the second argument, the function writes the file in the `do-while` loop. The read from `stdin` and the write to the file are done in the single statement that calls `writerecord()` with a call to `getdata()` as the first argument. The pointer to a `Record` object that `getdata()` returns is passed directly to the `writerecord()` function. The operation ends when the user enters `'n'` or `'N'` to indicate that no more data is to be entered. The file is closed before returning from the function.

Listing the File Contents

The prototype of a function that will list the records in a file on the standard output stream looks like this:

```
void listfile(char *filename);
```

The parameter is the name of the file, so the function will take care of opening the file initially and then closing it when the operation is complete.

Here's the implementation:

```
/* List the contents of the binary file */
void listfile(char *filename)
{
  /* Create the format string for names up to MAXLEN long */
  /* format array length allows up to 5 digits for MAXLEN */
  char format[15];                       /* Format string              */
  sprintf(format, "\n%%-%ds Age:%%4d", MAXLEN);

  FILE *pFile = fopen(filename, "rb");   /* Open file to read          */
  if(pFile == NULL)                      /* Check file is open         */
  {
    printf("Unable to open %s. Verify it exists.\n", filename);
    return;
  }

  struct Record record;                  /* Stores a record            */
  printf("\nThe contents of %s are:", filename);

  while(readrecord(&record, pFile) != NULL)  /* As long as we have records */
    printf(format, record.name, record.age); /* Output the record        */
```

```
  printf("\n");                              /* Move to next line          */

  fclose(pFile);                             /* Close the file             */
}
```

The function generates a format string that will adjust the name field width to be `MAXLEN` characters. The `sprintf()` function writes the format string to the format array.

The file is opened in binary read mode so the initial position will be at the beginning of the file. If the file is opened successfully, records are read from the file in the `while` loop by calling the `readrecord()` function that we defined earlier. The call to `readrecord()` is done in the loop condition so when `NULL` is returned signaling end-of-file has been detected, the loop ends. Within the loop you write the members of the `Record` object that was read by `readrecord()` to `stdout` using the string in the `format` array that was created initially. When all the records have been read, the file is closed by calling `fclose()` with the file pointer as the argument.

Updating the Existing File Contents

Updating existing records in the file adds a complication because of the variable length of the names in the file. You can't just arbitrarily overwrite an existing record because the chances are it won't fit in the space occupied by the record to be replaced. If the length of the new record is the same as the original, you can overwrite it. If they are different, the only solution is to write a new file. Here's the prototype of the function to update the file:

```
void updatefile(char *filename);
```

The only parameter is the file name, so the function will handle finding out which record is to be changed, as well as opening and closing the file. Here's the code:

```
/* Modify existing records in the file */
void updatefile(char *filename)
{  char answer = 'y';

  FILE *pFile = fopen(filename, "rb+");      /* Open the file for update   */
  if(pFile == NULL)                          /* Check file is open         */
  {
    fprintf(stderr, "\n File open for updating records failed.");
    return;
  }
  struct Record record;                      /* Stores a record            */
  int index = findrecord(&record, pFile);    /* Find the record for a name */
  if(index<0)                                /* If the record isn't there  */
  {
    printf("\nRecord not found.");           /* ouput a message            */
    return;                                  /* and we are done.           */
  }

  printf("\n%s is aged %d,", record.name, record.age);
  struct Record newrecord;                   /* Stores replacement record  */
  printf("\nYou can now enter the new name and age for %s.", record.name);
  getrecord(&newrecord);                     /* Get the new record         */
```

```
  /* Check if we can update in place */
  if((strlen(record.name) == strlen(newrecord.name)))
  { /* Name lengths are the same so we can */
    /* Move to start of old record         */
    fseek(pFile,
          -(long)(sizeof(size_t)+strlen(record.name)+sizeof(record.age)),
          SEEK_CUR);
    writerecord(&newrecord, pFile);          /* Write the new record       */
    fflush(pFile);                           /* Force the write            */
  }
  else
    duplicatefile(&newrecord, index, filename, pFile);

  printf("File update complete.\n");
}
```

There's quite a lot of code in this function but it consists of a sequence of fairly simple steps:

1. Open the file for update.
2. Find the index (first record at index 0) for the record to be updated.
3. Get the data for the record to replace the old record.
4. Check if the record can be updated in place. This is possible when the lengths of the names are the same. If so move the current position back by the length of the old record and write the new record to the old file.
5. If the names are different lengths duplicate the file with the new record replacing the old in the duplicate file.

After opening the file for update, the function reads the name corresponding to the record that is to be changed. The `findrecord()` function, which I'll get to in a moment, reads the name for the record to be updated, then returns the index value for that record, if it exists, with the first record at index 0. The `findrecord()` function will return –1 if the record is not found.

If the old and new names are the same length, move the file position back by the length of the old record by calling `fseek()`. Then write the new record to the file and flush the output buffer. Calling `fflush()` for the file forces the new record to be transferred from the file.

If the old and new records are different lengths, call `duplicatefile()` to copy the file with the new record replacing the old in the copy. You can implement the function like this:

```
/* Duplicate the existing file replacing the record to be update */
/* The record to be replaced is index records from the start     */
void duplicatefile(struct Record *pnewrecord, int index,
                                              char *filename, FILE *pFile)
{
    /* Create and open a new file */
  char tempname[L_tmpnam];
  if(tmpnam(tempname) == NULL)
  {
    printf("\nTemporary file name creation failed.");
    return;
  }
  char tempfile[strlen(dirpath)+strlen(tempname)+1];
  strcpy(tempfile, dirpath);                 /* Copy original file path  */
  strcat(tempfile, tempname);                /* Append temporary name    */
  FILE *ptempfile = fopen(tempfile, "wb+");
```

```
  /* Copy first index records from old file to new file */
  rewind(pFile);                          /* Old file back to start    */
  struct Record record;                   /* Store for a record        */
  for(int i = 0 ; i<index ; i++)
    writerecord(readrecord(&record, pFile), ptempfile);


  writerecord(pnewrecord, ptempfile);     /* Write the new record      */
  readrecord(&record,pFile);              /* Skip the old record       */

  /* Copy the rest of the old file to the new file */
  while(readrecord(&record,pFile))
    writerecord(&record, ptempfile);

  /* close the files */
  if(fclose(pFile)==EOF)
    printf("\n Failed to close %s", filename);
  if(fclose(ptempfile)==EOF)
    printf("\n Failed to close %s", tempfile);

  if(!remove(filename))                    /* Delete the old file       */
  {
    printf("\nRemoving the old file  failed. Check file in %s", dirpath);
    return;
  }

  /* Rename the new file same as original */
  if(!rename(tempfile, filename))
    printf("\nRenaming the file copy failed. Check file in %s", dirpath);
}
```

This function carries the update through the following steps:

1. Create a new file with a unique name in the same directory as the old file. The `dirpath` variable will be a global that contains the path to the original file.
2. Copy all records preceding the record to be changed from the old file to the new file.
3. Write the new record to the new file and skip over the record to be updated in the old file.
4. Write all the remaining records from the old file to the new file.
5. Close both files.
6. Delete the old file and rename the new file with the name of the old file.

Once the new file is created using the name generated by `tmpnam()`, records are copied from the original file to the new file, with the exception that the record to be updated is replaced with the new record in the new file. The copying of the first `index` records is done in the `for` loop where the pointer that is returned by `readrecord()` reading the old file is passed as the argument to `writerecord()` for the new file. The copying of the records that follow the updated record is done in the `while` loop. Here you have to continue copying records until the end-of-file is reached in the old file. Finally, after closing both files, delete the old file to free up its name and then rename the new file to the old. If you want to do this more safely, you can rename the old file in some way rather than deleting it, perhaps by appending `"_old"` to the existing file name. You can then rename the new file as you do here. This would leave a backup file in the directory that would be useful if the update goes awry.

The implementation of the `findrecord()` function that is called by `updatefile()` to find the index for the record that matches the name that is entered looks like this:

```
/* Find a record                          */
/* Returns the index number of the record */
/* or -1 if the record is not found.      */
int findrecord(struct Record *precord, FILE *pFile)
{
  char name[MAXLEN];
  printf("\nEnter the name for the record you wish to find: ");
  getname(name);

  rewind(pFile);                       /* Make sure we are at the start */
  int index = 0;                       /* Index of current record       */

  while(true)
  {
    readrecord(precord, pFile);
    if(feof(pFile))                     /* If end-of-file was reached    */
      return -1;                        /* record not found              */
    if(!strcmp(name, precord->name))
      break;
    ++index;
  }
  return index;                         /* Return record index           */
}
```

This function reads a name for the record that is to be changed, then reads records looking for a name that matches the name that was entered. If end-of-file is reached without finding the name, –1 is returned to signal to the calling program that the record is not in the file. If a name match is found, the function returns the index value of the matching record.

You can now assemble the complete working example.

TRY IT OUT: READING, WRITING, AND UPDATING A BINARY FILE

I won't repeat all the functions I have just described. You can add the following to the beginning of the source file containing the code for the functions other than `main()`:

```
/* Program 12.7 Writing, reading and updating a binary file */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

const int MAXLEN = 30;                          /* Size of name buffer     */
const char *dirpath = "C:\\temp\\";             /* Directory path for file */
const char *file = "mydata.bin";                /* File name               */

/* Structure encapsulating a name and age */
struct Record
{
```

```
  char name[MAXLEN];
  int age;
};

void listfile(char *filename);                    /* List the file contents   */
void updatefile(char *filename);                  /* Update the file contents */
struct Record *getrecord(struct Record *precord); /* Read a record from stdin */
void getname(char *pname);                        /* Read a name from stdin   */
void writefile(char *filename, char *mode);       /* Write records to a file  */
void writerecord(struct Record *precord, FILE *pFile);
struct Record * readrecord(struct Record *precord, FILE *pFile);
int findrecord(struct Record *precord, FILE *pFile);
void duplicatefile(struct Record *pnewrecord,
                                  int index, char *filename, FILE *pFile);
int main(void)
{
  char filename[strlen(dirpath)+strlen(file)+1];  /* Stores file path        */
  strcpy(filename, dirpath);                      /* Copy directory path     */
  strcat(filename, file);                         /*  and append file name   */

  /* Choose activity option */
  char answer = 'q';
  while(true)
  {
    printf("\nChoose from the following options:"
           "\nTo list the file contents enter  L"
           "\nTo create a new file enter       C"
           "\nTo add new records enter         A"
           "\nTo update existing records enter U"
           "\nTo delete the file enter         D"
           "\nTo end the program enter         Q\n : ");
    scanf("\n%c", &answer);

    switch(tolower(answer))
    {
      case 'l':                                  /* List file contents      */
        listfile(filename);
        break;
        case 'c':                                /* Create new file         */
        writefile(filename,"wb+");
        printf("\nFile creation complete.");
        break;
      case 'a':                                  /* Append records          */
        writefile(filename, "ab+");
        printf("\nFile append complete.");
        break;
      case 'u':                                  /* Update existing records */
        updatefile(filename);
        break;
      case 'd':
        printf("Are you sure you want to delete %s (y or n)? ", filename);
        scanf("\n%c", &answer);
        if(tolower(answer) == 'y')
          remove(filename);
```

```
        break;
      case 'q':                                /* Quit the program        */
        printf("\nEnding the program.", filename);
        return 0;
      default:
        printf("Invalid selection. Try again.");
        break;
    }
  }
  return 0;
}
```

Here's some sample output from a session using the main options offered by the program:

```
Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : c

Enter a name less than 30 characters:Bill Bloggs
Enter the age of Bill Bloggs: 22
Do you want to enter another(y or n)?  y

Enter a name less than 30 characters:Kitty Malone
Enter the age of Kitty Malone: 23
Do you want to enter another(y or n)?  n

File creation complete.
Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : l

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age:  22
Kitty Malone                   Age:  23

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : a

Enter a name less than 30 characters:Jack Flash
Enter the age of Jack Flash: 30
Do you want to enter another(y or n)?  n
```

```
File append complete.
Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : l

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age:  22
Kitty Malone                   Age:  23
Jack Flash                     Age:  30

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : u

Enter the name for the record you wish to find: Kitty Malone

Kitty Malone is aged 23,
You can now enter the new name and age for Kitty Malone.
Enter a name less than 30 characters:Kitty Moline
Enter the age of Kitty Moline: 24
File update complete.

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : l

The contents of C:\temp\mydata.bin are:
Bill Bloggs                    Age:  22
Kitty Moline                   Age:  24
Jack Flash                     Age:  30

Choose from the following options:
To list the file contents enter  L
To create a new file enter       C
To add new records enter         A
To update existing records enter U
To delete the file enter         D
To end the program enter         Q
 : q
```

How It Works

There's a lot of code in `main` but it's very simple. The global strings `dirpath` and `file` identify the directory and file name respectively for the file that contains the data. These are concatenated in `main()` with the result stored in `filename`.

The indefinite `while` loop offers a series of choices of action and the choice entered is determined in the switch statement. Depending on the character entered, one of the functions you developed for the program is called. Execution continues until the option `'Q'` or `'q'` is entered to end the program.

File Open Modes Summary

You probably will need a little practice before the file open mode strings come immediately to mind, so Table 12-2 contains a summary that you can refer back to when necessary.

Table 12-2. *File Modes*

Mode	Description
`"w"`	Open or create a text file for **w**rite operations.
`"a"`	Open a text file for **a**ppend operations, adding to the end of the file.
`"r"`	Open a text file for **r**ead operations.
`"wb"`	Open or create a **b**inary file for **w**rite operations.
`"ab"`	Open a **b**inary file for **a**ppend operations.
`"rb"`	Open a **b**inary file for **r**ead operations.
`"w+"`	Open or create a text file for update operations. An existing file will be truncated to zero length.
`"a+"`	Open or create a text file for update operations, adding to the end of the file.
`"r+"`	Open a text file for update operations (read and write anywhere).
`"w+b"` or `"wb+"`	Open or create a binary file for update operations. An existing file will be truncated to zero length.
`"a+b"` or `"ab+"`	Open a binary file for update operations, adding to the end of the file.
`"r+b"` or `"rb+"`	Open a binary file for update operations (read and write anywhere).

Designing a Program

Now that you've come to the end of this chapter, you can put what you've learned into practice with a final program. This program will be shorter than the previous example, but nonetheless it's an interesting program that you may find useful.

The Problem

The problem you're going to solve is to write a file-viewer program. This will display any file in hexadecimal representation and as characters.

The Analysis

The program will open the file as binary read-only and then display the information in two columns, the first being the hexadecimal representation of the bytes in the file, and the second being the bytes represented as characters. The file name will be supplied as a command line argument or, if that isn't supplied, the program will ask for the file name.

The stages are as follows:

1. If the file name isn't supplied, get it from the user.
2. Open the file.
3. Read and display the contents of the file.

The Solution

This section outlines the steps you'll take to solve the problem.

Step 1

You can easily check to see if the file name appears at the command line by specifying that the function `main()` has parameters. Up until now, we have ignored the possibility of parameters being passed to `main()`, but here you can use it as the means of identifying the file that's to be displayed. You'll recall that when `main()` is called, two parameters are passed to it. The first parameter is an integer indicating the number of words in the command line, and the second is an array of pointers to strings. The first string contains the name that you use to start the program at the command line, and the remaining strings represent the arguments that follow at the command line. Of course, this mechanism allows an arbitrary number of values to be entered at the command line and passed to `main()`.

If the value of the first argument to `main()` is 1, there's only the program name on the command line, so in this case you'll have to prompt for the file name to be entered:

```
/* Program 12.8 Viewing the contents of a file */
#include <stdio.h>

const int MAXLEN = 256;                    /* Maximum file path length      */

int main(int argc, char *argv[])
{
  char filename[MAXLEN];                   /* Stores the file path          */

  if(argc == 1)                            /* No file name on command line? */
  {
    printf("Please enter a filename: ");   /* Prompt for input              */
    fgets(filename, MAXLEN, stdin);        /* Get the file name entered     */
```

```
    /* Remove the newline if it's there */
    int len = strlen(filename);
    if(filename[len-1] == '\n')
      filename[len-1] = '\0';
  }
  return 0;
}
```

This allows for a maximum file path length of 256 characters.

Step 2

If the first argument to `main()` isn't 1, then you have at least one more argument, which you assume is the file name. You therefore copy the string pointed to by `argv[1]` to the variable `openfile`. Assuming that you have a valid file name, you can open the file and start reading it:

```
/* Program 12.8 Viewing the contents of a file */
#include <stdio.h>

const int MAXLEN = 256;                    /* Maximum file path length      */

int main(int argc, char *argv[])
{
  char filename[MAXLEN];                   /* Stores the file path          */
  FILE *pfile;                             /* File pointer                  */

  if(argc == 1)                            /* No file name on command line? */
  {
    printf("Please enter a filename: ");   /* Prompt for input              */
    fgets(filename, MAXLEN, stdin);        /* Get the file name entered     */

    /* Remove the newline if it's there */
    int len = strlen(filename);
    if(filename[len-1] == '\n')
      filename[len-1] = '\0';
  }
  else
    strcpy(filename, argv[1]);             /* Get 2nd command line string   */

  /* File can be opened OK? */
  if(!(pfile = fopen(filename, "rb")))
  {
    printf("Sorry, can't open %s", filename);
    return -1;
  }
  fclose(pfile);                           /* Close the file                */
  return 0;
}
```

You put the call to the `fclose()` function to close the file at the end of the program so that you don't forget about it later. Also, you use a return value of –1 for the program to indicate when an error has occurred.

Step 3

You can now output the file contents. You do this by reading the file one byte at a time and saving this data in a buffer. Once the buffer is full or the end of file has been reached, you output the buffer in the format you want. When you output the data as characters, you must first check that the character is printable, otherwise strange things may start happening on the screen. You use the function `isprint()`, declared in `ctype.h`, for this. If the character isn't printable, you'll print a period instead.

Here's the complete code for the program:

```
/* Program 12.8 Viewing the contents of a file */
#include <stdio.h>
#include <ctype.h>
#include <string.h>

const int MAXLEN = 256;                  /* Maximum file path length      */
const int DISPLAY = 80;                  /* Length of display line        */
const int PAGE_LENGTH = 20;              /* Lines per page                */

int main(int argc, char *argv[])
{
  char filename[MAXLEN];                 /* Stores the file path          */
  FILE *pfile;                           /* File pointer                  */
  unsigned char buffer[DISPLAY/4 - 1];   /* File input buffer             */
  int count = 0;                         /* Count of characters in buffer */
  int lines = 0;                         /* Number of lines displayed     */

  if(argc == 1)                          /* No file name on command line? */
  {
    printf("Please enter a filename: "); /* Prompt for input              */
    fgets(filename, MAXLEN, stdin);      /* Get the file name entered     */

    /* Remove the newline if it's there */
    int len = strlen(filename);
    if(filename[len-1] == '\n')
      filename[len-1] = '\0';
  }
  else
    strcpy(filename, argv[1]);           /* Get 2nd command line string   */

    /* File can be opened OK?        */
    if(!(pfile = fopen(filename, "rb")))
    {
      printf("Sorry, can't open %s", filename);
      return -1;
    }
  while(!feof(pfile))                    /* Continue until end of file    */
  {
    if(count < sizeof buffer)            /* If the buffer is not full     */
      buffer[count++] = (unsigned char)fgetc(pfile);    /* Read a character */
    else
    { /* Output the buffer contents, first as hexadecimal */
      for(count = 0; count < sizeof buffer; count++)
        printf("%02X ", buffer[count]);
      printf("| ");                       /* Output separator              */
```

```
      /* Now display buffer contents as characters */
      for(count = 0; count < sizeof buffer; count++)
        printf("%c", isprint(buffer[count]) ? buffer[count]:'.');
      printf("\n");                          /* End the line                */
      count = 0;                             /* Reset count                 */

      if(!(++lines%PAGE_LENGTH))             /* End of page?                */
       if(getchar()=='E')                    /* Wait for Enter              */
         return 0;                           /* E pressed                   */
    }
  }

  /* Display the last line, first as hexadecimal */
  for(int i = 0; i < sizeof buffer; i++)
    if(i < count)
      printf("%02X ", buffer[i]);            /* Output hexadecimal          */
    else
      printf("   ");                         /* Output spaces               */
  printf("| ");                              /* Output separator            */

  /* Display last line as characters */
  for(int i = 0; i < count; i++)
    /* Output character    */
    printf("%c",isprint(buffer[i]) ? buffer[i]:'.');

  /* End the line */
  printf("\n");
  fclose(pfile);                             /* Close the file              */
  return 0;
}
```

The symbol `DISPLAY` specifies the width of a line on the screen for output, and the symbol `PAGE_LENGTH` specifies the number of lines per page. You arrange to display a page, and then wait for Enter to be pressed before displaying the next page, thus avoiding the whole file whizzing by before you can read it.

You declare the buffer to hold input from the file as

```
unsigned char buffer[DISPLAY/4 - 1];     /* File input buffer           */
```

The expression for the array dimension arises from the fact that you'll need four characters on the screen to display each character from the file, plus one separator. Each character will be displayed as two hexadecimal digits plus a space, and as a single character, making four characters in all.

You continue reading as long as the `while` loop condition is true:

```
while(!feof(pfile))                      /* Continue until end of file  */
```

The library function, `feof()`, returns `true` if `EOF` is read from the file specified by the argument; otherwise, it returns `false`.

You fill the `buffer` array with characters from the file in the `if` statement:

```
    if(count < sizeof buffer)            /* If the buffer is not full     */
      buffer[count++] = (unsigned char)fgetc(pfile);   /* Read a character  */
```

When `count` exceeds the capacity of `buffer`, the `else` clause will be executed to output the contents of the buffer:

```
else
{ /* Output the buffer contents, first as hexadecimal */
  for(count = 0; count < sizeof buffer; count++)
    printf("%02X ", buffer[count]);
  printf("| ");                          /* Output separator              */

  /* Now display buffer contents as characters */
  for(count = 0; count < sizeof buffer; count++)
    printf("%c", isprint(buffer[count]) ? buffer[count]:'.');
  printf("\n");                          /* End the line                  */
  count = 0;                             /* Reset count                   */

  if(!(++lines%PAGE_LENGTH))             /* End of page?                  */
   if(getchar()=='E')                    /* Wait for Enter                */
     return 0;                           /* E pressed                     */
}
```

The first `for` loop outputs the contents of `buffer` as hexadecimal characters. You then output a separator character and execute the next `for` loop to output the same data as characters. The conditional operator in the second argument to `printf()` ensures that nonprinting characters are output as a period.

The `if` statement increments the line count, `lines`, and for every `PAGE_LENGTH` number of lines, wait for a character to be entered. If you press Enter, the next pageful will be displayed, but if you press **E** and then Enter, the program will end. This provides you with an opportunity to escape from continuing to output the contents of a file that's larger than you thought.

The final couple of `for` loops are similar to those you've just seen. The only difference is that spaces are output for array elements that don't contain file characters. An example of the output is as follows. It shows part of the source file for the self-same program and you can deduce from the output that the file path was entered as a command line argument.

```
2F 2A 20 50 72 6F 67 72 61 6D 20 31 32 2E 38 20 56 69 65 | /* Program 12.8 Vie
77 69 6E 67 20 74 68 65 20 63 6F 6E 74 65 6E 74 73 20 6F | wing the contents o
66 20 61 20 66 69 6C 65 20 2A 2F 0D 0A 23 69 6E 63 6C 75 | f a file */..#inclu
64 65 20 3C 73 74 64 69 6F 2E 68 3E 0D 0A 23 69 6E 63 6C | de <stdio.h>..#incl
75 64 65 20 3C 63 74 79 70 65 2E 68 3E 0D 0A 23 69 6E 63 | ude <ctype.h>..#inc
6C 75 64 65 20 3C 73 74 72 69 6E 67 2E 68 3E 0D 0A 0D 0A | lude <string.h>....
63 6F 6E 73 74 20 69 6E 74 20 4D 41 58 4C 45 4E 20 3D 20 | const int MAXLEN =
32 35 36 3B 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | 256;
20 20 20 20 20 2F 2A 20 4D 61 78 69 6D 75 6D 20 66 69 6C | /* Maximum fil
65 20 70 61 74 68 20 6C 65 6E 67 74 68 20 20 20 20 20 20 | e path length
2A 2F 0D 0A 63 6F 6E 73 74 20 69 6E 74 20 44 49 53 50 4C | */..const int DISPL
41 59 20 3D 20 38 30 3B 20 20 20 20 20 20 20 20 20 20 20 | AY = 80;
20 20 20 20 20 20 20 20 20 2F 2A 20 4C 65 6E 67 74 68 20 | /* Length
6F 66 20 64 69 73 70 6C 61 79 20 6C 69 6E 65 20 20 20 20 | of display line
20 20 20 20 2A 2F 0D 0A 63 6F 6E 73 74 20 69 6E 74 20 50 | */..const int P
41 47 45 5F 4C 45 4E 47 54 48 20 3D 20 32 30 3B 20 20 20 | AGE_LENGTH = 20;
20 20 20 20 20 20 20 20 20 20 20 20 20 2F 2A 20 4C 69 6E | /* Lin
65 73 20 70 65 72 20 70 61 67 65 20 20 20 20 20 20 20 20 | es per page
20 20 20 20 20 20 20 20 2A 2F 0D 0A 0D 0A 69 6E 74 20 6D | */....int m
61 69 6E 28 69 6E 74 20 61 72 67 63 2C 20 63 68 61 72 20 | ain(int argc, char
```

A lot more output follows, ending with this:

```
66 65 72 5B 69 5D 3A 27 2E 27 29 3B 0D 0A 20 20 2F 2A 20 | fer[i]:'.');..  /*
45 6E 64 20 74 68 65 20 6C 69 6E 65 20 20 20 20 20 20 20 | End the line
20 20 20 2A 2F 0D 0A 20 20 70 72 69 6E 74 66 28 22 5C 6E |    */..  printf("\n
22 29 3B 0D 0A 20 20 66 63 6C 6F 73 65 28 70 66 69 6C 65 | ");..  fclose(pfile
29 3B 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 | );
20 20 20 20 20 20 20 20 20 20 2F 2A 20 43 6C 6F 73 65 20 |           /* Close
74 68 65 20 66 69 6C 65 20 20 20 20 20 20 20 20 20 20 20 | the file
20 20 20 20 20 2A 2F 0D 0A 20 20 72 65 74 75 72 6E 20 30 |      */..  return 0
3B 0D 0A 7D 0D 0A 0D 0A FF                               | ;..}.....
```

Summary

Within this chapter I've covered all of the basic tools necessary to provide you with the ability to program the complete spectrum of file functions. The degree to which these have been demonstrated in examples has been, of necessity, relatively limited. There are many ways of applying these tools to provide more sophisticated ways of managing and retrieving information in a file. For example, it's possible to write index information into the file, either as a specific index at a known place in the file, often the beginning, or as position pointers within the blocks of data, rather like the pointers in a linked list. You should experiment with file operations until you feel confident that you understand the mechanisms involved.

Although the functions I discussed in this chapter cover most of the abilities you're likely to need, you'll find that the input/output library provided with your compiler offers quite a few additional functions that give you even more options for handling your file operations.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 12-1. Write a program that will write an arbitrary number of strings to a file. The strings should be entered from the keyboard and the program shouldn't delete the file, as it will be used in the next exercise.

Exercise 12-2. Write a program that will read the file that was created by the previous exercise, and retrieve the strings one at a time in reverse sequence and write them to a new file in the sequence in which they were retrieved. For example, the program will retrieve the last string and write that to the new file, then retrieve the second to last and retrieve that from the file, and so on, for each string in the original file.

Exercise 12-3. Write a program that will read names and telephone numbers from the keyboard and write them to a new file if a file doesn't already exist and add them if the file does exist. The program should optionally list all the entries.

Exercise 12-4. Extend the program from the previous exercise to implement retrieval of all the numbers corresponding to a given second name. The program should allow further enquiries, adding new name/number entries and deleting existing entries.

CHAPTER 13

Supporting Facilities

At this point you've covered the complete C language, as well as the important library functions. You should be reasonably confident in programming all aspects of the language. If you aren't, that's simply because you need more practice. Once you've learned the elements of the language, competence comes down to practice, practice, practice.

In this final chapter, I'll tie up a few loose ends. You'll delve deeper into the capabilities you have available through the preprocessor, and you'll look at a few more library functions that you're sure to find useful.

In this chapter you'll learn the following:

- More about the preprocessor and its operation
- How to write preprocessor macros
- What logical preprocessor directives are and how you can use them
- What conditional compilation is and how you can apply it
- More about the debugging methods that are available to you
- How you use some of the additional library functions available

Preprocessing

As you are certainly aware by now, preprocessing of your source code occurs before it's compiled to machine instructions. The preprocessing phase can execute a range of service operations specified by preprocessing directives, which are identified by the # symbol as the first character of each preprocessor directive. The preprocessing phase provides an opportunity for manipulating and modifying your C source code prior to compilation. Once the preprocessing phase is complete and all directives have been analyzed and executed, all such preprocessing directives will no longer appear in the source code that results. The compiler begins the compile phase proper, which generates the machine code equivalent of your program.

You've already used preprocessor directives in all the examples, and you're familiar with both the `#include` and `#define` directives. There are a number of other directives that add considerable flexibility to the way in which you specify your programs. Keep in mind as you proceed that all these are preprocessing operations that occur before your program is compiled. They modify the set of statements that constitute your program. They aren't involved in the execution of your program at all.

Including Header Files in Your Programs

A header file is any external file whose contents are included into your program by use of the `#include` preprocessor directive. You're completely familiar with statements such as this:

```
#include <stdio.h>
```

This fetches the standard library header file supporting input/output operations into your program. This is a particular case of the general statement for including standard libraries into your program:

```
#include <standard_library_file_name>
```

Any library header file name can appear between the angled brackets. If you include a header file that you don't use, the only effect, apart from slightly confusing anyone reading the program, is to extend the compilation time.

You can include your own source files into your program with a slightly different `#include` statement. A typical example might be this:

```
#include "myfile.h"
```

This statement will introduce the contents of the file named between double quotes into the program in place of the `#include` directive. The contents of any file can be included into your program by this means. You simply specify the name of the file between quotes as shown in the example.

You can also give the file whatever name you like within the constraints of the operating system. In theory you don't have to use the extension `.h`, although it's a convention commonly adhered to by most programmers in C, so I strongly recommend that you use it too. The difference between using this form and using angled brackets lies in the source that will be assumed for the required file. The precise operation is compiler-dependent and will be described in the compiler documentation, but usually the first form will search the default header file directory for the required file, whereas the second form will search the current source directory before the default header file directory is searched.

Header files cannot include implementation, by which I mean executable code. You use header files to contain declarations, not function definitions or initialized global data. All your function definitions and initialized globals are placed in source files with the extension `.c`. You can use the header file mechanism to divide your program source code into several files and, of course, to manage the declarations for any library functions of your own. A very common use of this facility is to create a header file containing all the function prototypes and type declarations for a program. These can then be managed as a separate unit and included at the beginning of any source file for the program. You need to avoid duplicating information if you include more than one header file into a source file. Duplicate code will often cause compilation errors. You'll see later in this chapter in the "Conditional Compilation" section how you can ensure that any given block of code will appear only once in your program, even if you inadvertently include it several times.

■ **Note** A file introduced into your program by an `#include` directive may also contain another `#include` directive. If so, preprocessing will deal with the second `#include` in the same way as the first and continue replacing such directives with the contents of the corresponding file until there are no more `#include` directives in the program.

External Variables and Functions

With a program that's made up of several source files, you'll frequently find that you want to use a global variable that's defined in another file. You can do this by declaring the variable as external to

the current file using the `extern` keyword. For example, if you have variables defined in another file as **global** (which means outside of any of the functions) using the statements

```
int number = 0;
double in_to_mm = 2.54;
```

then in a function in which you want to access these, you can specify that these variable names are external by using these statements:

```
extern int number;
extern double in_to_mm;
```

These statements don't create these variables—they just identify to the compiler that these names are defined elsewhere, and this assumption about these names should apply to the rest of this source file. The variables you specify as `extern` must be declared and defined somewhere else in the program, usually in another source file. If you want to make these external variables accessible to all functions within the current file, you should declare them as external at the very beginning of the file, prior to any of the function definitions. With programs consisting of several files, you could place all initialized global variables at the beginning of one file and all the `extern` statements in a header file. The `extern` statements can then be incorporated into any program file that needs access to these variables by using an `include` statement for the header file.

■ **Note** Only one declaration of each global variable is allowed in a file. Of course, the global variables may be declared as external in as many files as necessary.

Substitutions in Your Program Source Code

There are preprocessor directives for replacing symbols in your source code before it is compiled. The simplest kind of symbol substitution you can make is one you've already seen. For example, the preprocessor directive to substitute the specified numeric value, wherever the character string `PI` occurs, is as follows:

```
#define PI 3.14159265
```

Although the identifier `PI` *looks* like a variable, it is not a variable and has nothing to do with variables. Here `PI` is a token, rather like a voucher, that is exchanged for the sequence of digits specified in the `#define` directive during the preprocessing phase. When your program is ready to be compiled after preprocessing has been completed, the string `PI` will no longer appear, having been replaced by its definition wherever it occurs in the source file. The general form of this sort of preprocessor directive is the following:

```
#define identifier sequence_of_characters
```

Here, `identifier` conforms to the usual definition of an identifier in C: any sequence of letters and digits, the first of which is a letter, and underline characters count as letters. Note that `sequence_of_characters`, which is the replacement for `identifier`, is any sequence of characters and need not be just digits.

A very common use of the `#define` directive is to define array dimensions by way of a substitution, to allow a number of array dimensions to be determined by a single token. Only one directive in the program then needs to be modified to alter the dimensions of a number of arrays in the program. This helps considerably in minimizing errors when such changes are necessary, as shown in the following example:

```
#define MAXLEN 256
char *buffer[MAXLEN];
char *str[MAXLEN];
```

Here, the dimensions of both arrays can be changed by modifying the single `#define` directive, and of course the array declarations that are affected can be anywhere in the program file. The advantages of this approach in a large program involving dozens or even hundreds of functions should be obvious. Not only is it easy to make a change, but also using this approach ensures that the same value is being used through a program. This is especially important with large projects involving several programmers working together to produce the final product.

Of course, you can also define a value such as `MAXLEN` as a `const` variable:

```
const size_t MAXLEN = 256;
```

The difference between this approach and using the `#define` directive is that `MAXLEN` here is no longer a token but is a variable of a specific type with the name `MAXLEN`. The `MAXLEN` in the `#define` directive does not exist once the source file has been preprocessed because all occurrences of `MAXLEN` in the code will be replaced by `256`.

I use numerical substitution in the last two examples, but as I said, you're in no way limited to this. You could, for example, write the following:

```
#define Black White
```

This will cause any occurrence of `Black` in your program to be replaced with `White`. The sequence of characters that is to replace the token identifier can be anything at all.

Macro Substitutions

A **macro** is based on the ideas implicit in the `#define` directive examples you've seen so far, but it provides a greater range of possible results by allowing what might be called **multiple parameterized substitutions**. This not only involves substitution of a fixed sequence of characters for a token identifier, but also allows parameters to be specified, which may themselves be replaced by argument values, wherever the parameter appears in the substitution sequence.

Let's look at an example:

```
#define Print(My_var) printf("%d", Myvar)
```

This directive provides for two levels of substitution. There is the substitution for `Print(My_var)` by the string immediately following it in the `#define` statement, and there is the possible substitution of alternatives for `My_var`. You could, for example, write the following:

```
Print(ival);
```

This will be converted during preprocessing to this statement:

```
printf("%d", ival);
```

You could use this directive to specify a `printf()` statement for an integer variable at various points in your program. A common use for this kind of macro is to allow a simple representation of a complicated function call in order to enhance the readability of a program.

Macros That Look Like Functions

The general form of the kind of substitution directive just discussed is the following:

```
#define macro_name( list_of_identifiers ) substitution_string
```

This shows that in the general case, multiple parameters are permitted, so you're able to define more complex substitutions.

To illustrate how you use this, you can define a macro for producing a maximum of two values with the following directive:

```
#define max(x, y) x>y ? x : y
```

You can then put the statement in the program:

```
result = max(myval, 99);
```

This will be expanded during preprocessing to produce the following code:

```
result = myval>99 ? myval : 99;
```

It's important to be conscious of the substitution that is taking place and not to assume that this is a function. You can get some strange results otherwise, particularly if your substitution identifiers include an explicit or implicit assignment. For example, the following modest extension of the last example can produce an erroneous result:

```
result = max(myval++, 99);
```

The substitution process will generate this statement:

```
result = myval++>99 ? myval++ : 99;
```

The consequence of this is that if the value of `myval` is larger than 99, `myval` will be incremented twice. Note that it does *not* help to use parentheses in this situation. If you write the statement as

```
result = max((myval++), 99);
```

preprocessing will convert this to

```
result = (myval++>99) ? (myval++) : 99;
```

You need to be very cautious if you're writing macros that generate expressions of any kind. In addition to the multiple substitution trap you've just seen, precedence rules can also catch you out. A simple example will illustrate this. Suppose you write a macro for the product of two parameters:

```
#define product(m, n)  m*n
```

You then try to use this macro with the following statement:

```
result = product(x, y + 1);
```

Of course, everything works fine so far as the macro substitution is concerned, but you don't get the result you want, as the macro expands to this:

```
result = x*y + 1;
```

It could take a long time to discover that you aren't getting the product of the two parameters at all in this case, as there's no external indication of what's going on. There's just a more or less erroneous value propagating through the program. The solution is very simple. If you use macros to generate expressions, put parentheses around everything. So you should rewrite the example as follows:

```
#define product(m, n)  ((m)*(n))
```

Now everything will work as it should. The inclusion of the outer parentheses may seem excessive, but because you don't know the context in which the macro expansion will be placed, it's better to include them. If you write a macro to sum its parameters, you will easily see that without the outer

parentheses, there are many contexts in which you will get a result that's different from what you expect. Even with parentheses, expanded expressions that repeat a parameter, such as the one you saw earlier that uses the conditional operator, will still not work properly when the argument involves the increment or decrement operators.

Preprocessor Directives on Multiple Lines

A preprocessor directive must be a single logical line, but this doesn't prevent you from using the statement continuation character, \.

You could write the following:

```
#define min(x, y) \
              ((x)<(y) ? (x) : (y))
```

Here, the directive definition continues on the second line with the first nonblank character found, so you can position the text on the second line wherever you feel looks like the nicest arrangement. Note that the \ must be the last character on the line, immediately before you press Enter.

Strings As Macro Arguments

String constants are a potential source of confusion when used with macros. The simplest string substitution is a single-level definition such as the following:

```
#define MYSTR "This string"
```

Suppose you now write the statement

```
printf("%s", MYSTR);
```

This will be converted during preprocessing into the statement

```
printf("%s", "This string");
```

This should be what you are expecting. You couldn't use the `#define` directive without the quotes in the substitution sequence and expect to be able to put the quotes in your program text instead. For example, suppose you write the following:

```
#define  MYSTR  This string
  ...
printf("%s", "MYSTR" );
```

There will be no substitution for `MYSTR` in the `printf()` function in this case. Anything in quotes in your program is assumed to be a literal string, so it won't be analyzed during preprocessing.

There's a special way of specifying that the substitution for a macro argument is to be implemented as a string. For example, you could specify a macro to display a string using the function `printf()` as follows:

```
#define PrintStr(arg) printf("%s", #arg)
```

The # character preceding the appearance of the `arg` parameter in the macro expansion indicates that the argument is to be surrounded by double quotes when the substitution is generated. Therefore, if you write the following statement in your program

```
PrintStr(Output);
```

it will be converted during preprocessing to

```
printf("%s", "Output");
```

You may be wondering why this apparent complication has been introduced into preprocessing. Well, without this facility you wouldn't be able to include a variable string in a macro definition at all. If you were to put the double quotes around the macro parameter, it wouldn't be interpreted as a variable; it would be merely a string with quotes around it. On the other hand, if you put the quotes in the macro expansion, the string between the quotes wouldn't be interpreted as an identifier for a parameter; it would be just a string constant. So what might appear to be an unnecessary complication at first sight is actually an essential tool for creating macros that allow strings between quotes to be created.

A common use of this mechanism is for converting a variable name to a string, such as in this directive:

```
#define show(var) printf("\n%s = %d", #var, var);
```

If you now write

```
show(number);
```

this will generate the statement

```
printf("\n%s = %d", "number", number);
```

You can also generate a substitution that would allow you to display a string with double quotes included. Assuming you've defined the macro `PrintStr` as shown previously and you write the statement

```
PrintStr("Output");
```

it will be preprocessed into the statement

```
printf("%s", "\"Output\"");
```

This is possible because the preprocessing phase is clever enough to recognize the need to put `\"` at each end to get a string that includes double quotes to be displayed correctly.

Joining Two Results of a Macro Expansion

There are times when you may wish to generate two results in a macro and join them together with no spaces between them. Suppose you try to define a macro to do this as follows:

```
#define join(a, b) ab
```

This can't work in the way you need it to. The definition of the expansion will be interpreted as `ab`, not as the parameter `a` followed by the parameter `b`. If you separate them with a blank, the result will be separated with a blank, which isn't what you want either. The preprocessing phase provides you with another operator to solve this problem. The solution is to specify the macro as follows:

```
#define join(a, b) a##b
```

The presence of the operator consisting of the two characters ## serves to separate the parameters and to indicate that the result of the two substitutions are to be joined. For example, writing the statement

```
strlen(join(var, 123));
```

will result in the statement

```
strlen(var123);
```

This might be applied to synthesizing a variable name for some reason, or generating a format control string from two or more macro parameters.

Logical Preprocessor Directives

The last example you looked at appears to be of limited value, because it's hard to envision when you would want to simply join `var` to `123`. After all, you could always use one parameter and write `var123` as the argument. One aspect of preprocessing that adds considerably more potential to the previous example is the possibility for multiple macro substitutions where the arguments for one macro are derived from substitutions defined in another. In the last example, both arguments to the `join()` macro could be generated by other `#define` substitutions or macros. Preprocessing also supports directives that provide a logical `if` capability, which vastly expands the scope of what you can do during the preprocessing phase.

Conditional Compilation

The first logical directive I'll discuss allows you to test whether an identifier exists as a result of having been created in a previous `#define` directive. It takes the following form:

```
#if defined identifier
```

If the specified `identifier` is defined, statements that follow the `#if` are included in the program code until the directive `#endif` is reached. If the identifier isn't defined, the statements between the `#if` and the `#endif` will be skipped. This is the same logical process you use in C programming, except that here you're applying it to the inclusion or exclusion of program statements in the source file.

You can also test for the absence of an identifier. In fact, this tends to be used more frequently than the form you've just seen. The general form of this directive is:

```
#if !defined identifier
```

Here, the statements following the `#if` down to the `#endif` will be included if the identifier hasn't previously been defined. This provides you with a method of avoiding duplicating functions, or other blocks of code and directives, in a program consisting of several files, or to ensure bits of code that may occur repeatedly in different libraries aren't repeated when the `#include` statements in your program are processed.

The mechanism is simply to top and tail the block of code you want to avoid duplicating as follows:

```
#if !defined block1
  #define block1
  /* Block of code you do not */
  /* want to be repeated.     */
#endif
```

If the identifier `block1` hasn't been defined, the block following the `#if` will be included and processed, and `block1` will be defined. The following block of code down to the `#endif` will also be included in your program. Any subsequent occurrence of the same group of statements won't be included because the identifier `block1` now exists.

The `#define` directive doesn't need to specify a substitution value in this case. For the conditional directives to operate, it's sufficient for `block1` to appear in a `#define` directive. You can now include this block of code anywhere you think you might need it, with the assurance that it will never be duplicated within a program. The preprocessing directives ensure this can't happen.

■ **Note** It's a good idea to get into the habit of always protecting code in your own libraries in this fashion. You'll be surprised how easy it is to end up duplicating blocks of code accidentally, once you've collected a few libraries of your own functions.

You aren't limited to testing just one value with the `#if` preprocessor directive. You can use logical operators to test if multiple identifiers have been defined. For example, the statement

```
#if defined block1 && defined block2
```

will evaluate to `true` if both `block1` and `block2` have previously been defined, and so the code that follows such a directive won't be included unless this is the case.

A further extension of the flexibility in applying the conditional preprocessor directives is the ability to undefine an identifier you've previously defined. This is achieved using a directive such as `#undef block1`.

Now, if `block1` was previously defined, it is no longer defined after this directive. The ways in which these directives can all be combined to useful effect is only limited by your own ingenuity.

There are alternative ways of writing these directives that are slightly more concise. You can use whichever of the following forms you prefer. The directive `#ifdef block` is the same as `#if defined block`. And the directive `#ifndef block` is the same as `#if !defined block`.

Directives Testing for Specific Values

You can also use a form of the `#if` directive to test the value of a constant expression. If the value of the constant expression is nonzero, the following statements down to the next `#endif` are included in the program code. If the constant expression evaluates to zero, the following statements down to the next `#endif` are skipped. The general form of the `#if` directive is the following:

```
#if constant_expression
```

This is most frequently applied to test for a specific value being assigned to an identifier by a previous preprocessing directive. You might have the following sequence of statements, for example:

```
#if CPU == Pentium4
  printf("\nPerformance should be good." );
#endif
```

The `printf()` statement will be included in the program here only if the identifier `CPU` has been defined as `Pentium4` in a previous `#define` directive.

Multiple-Choice Selections

To complement the `#if` directives, you have the `#else` directive. This works in exactly the same way as the `else` statement does, in that it identifies a group of directives to be executed or statements to be included if the `#if` condition fails, for example:

```
#if CPU == Pentium4
  printf("\nPerformance should be good." );
#else
  printf("\nPerformance may not be so good." );
#endif
```

In this case, one or the other of the `printf()` statements will be included, depending on whether `CPU` has been defined as `Pentium4`.

The preprocessing phase also supports a special form of the #if for multiple-choice selections, in which only one of several choices of statements for inclusion in the program is required. This is the #elif directive, which has the general form

```
#elif constant_expression
```

An example of using this would be as follows:

```
#define US 0
#define UK 1
#define Australia 2
#define Country US
#if Country == US
  #define Greeting "Howdy, stranger."
#elif Country == UK
  #define Greeting "Wotcher, mate."
#elif Country == Australia
  #define Greeting "G'day, sport."
#endif
printf("\n%s", Greeting );
```

With this sequence of directives the output of the printf() statement will depend on the value assigned to the identifier Country, in this case US.

Standard Preprocessing Macros

There are usually a considerable number of standard preprocessing macros defined, which you'll find described in your compiler documentation. I'll just mention two that are of general interest and that are available to you.

The __DATE__ macro provides a string representation of the date in the form Mmm dd yyyy when it's invoked in your program. Here Mmm is the month in characters, such as Jan, Feb, and so on. The pair of characters dd is the day in the form of a pair of digits 1 to 31, where single-digit days are preceded by a blank. Finally, yyyy is the year as four digits—2006, for example.

A similar macro, __TIME__, provides a string containing the value of the time when it's invoked, in the form hh:mm:ss, which is evidently a string containing pairs of digits for hours, minutes, and seconds, separated by colons. Note that the time is when the compiler is executed, not when the program is run.

You could use this macro to record when your program was last compiled with a statement such as this:

```
printf("\nProgram last compiled at %s on %s", __TIME__, __DATE__ );
```

Note that both __DATE__ and __TIME__ have two underscore characters at the beginning and the end. Once the program containing this statement is compiled, the values that will be output by the printf() statement are fixed until you compile it again. On subsequent executions of the program, the then current time and date will be output. Don't confuse these macros with the time function I discuss later in this chapter in the section "The Date and Time Function Library."

Debugging Methods

Most of your programs will contain errors, or **bugs**, when you first complete them. Removing such bugs from a program can represent a substantial proportion of the time required to write the program. The larger and more complex the program, the more bugs it's likely to contain, and the more time it

will take to get the program to run properly. Very large programs, such as those typified by operating systems, or complex applications, such as word processing systems or even C program development systems, can be so involved that all the bugs can never be eliminated. You may already have experience of this in practice with some of the systems on your own computer. Usually these kinds of residual bugs are relatively minor, with ways in the system to work around them.

Your approach to writing a program can significantly affect how difficult it will be to test. A well-structured program consisting of compact functions, each with a well-defined purpose, is much easier to test than one without these attributes. Finding bugs will also be easier in a program that has extensive comments documenting the operation and purpose of its component functions and has well-chosen variable and function names. Good use of indentation and statement layout can also make testing and fault finding simpler.

It's beyond the scope of this book to deal with debugging comprehensively, but in this section I introduce the basic ideas that you need to be aware of.

Integrated Debuggers

Many compilers are supplied with extensive debugging tools built into the program development environment. These can be very powerful facilities that can dramatically reduce the time required to get a program working. They typically provide a varied range of aids to testing a program that include the following:

Tracing program flow: This capability allows you to execute your program one source statement at a time. It operates by pausing execution after each statement and continuing with the next statement after you press a designated key. Other provisions of the debug environment will usually allow you to display information easily, pausing to show you what's happening to the data in your program.

Setting breakpoints: Executing a large or complex program one statement at a time can be very tedious. It may even be impossible in a reasonable period of time. All you need is a loop that executes 10,000 times to make it an unrealistic proposition. Breakpoints provide an excellent alternative. With breakpoints, you define specific selected statements in your program at which a pause should occur to allow you to check what's happening. Execution continues to the next breakpoint when you press a specified key.

Setting watches: This sort of facility allows you to identify variables that you want to track the value of as execution progresses. The values of the variables that you select are displayed at each pause point in your program. If you step through your program statement by statement, you can see the exact point at which values are changed, or perhaps not changed when you expect them to be.

Inspecting program elements: It may also be possible to examine a wide variety of program components. For example, at breakpoints the inspection can show details of a function such as its return type and its arguments. You can also see details of a pointer in terms of its address, the address it contains, and the data stored at the address contained in the pointer. Seeing the values of expressions and modifying variables may also be provided for. Modifying variables can help to bypass problem areas to allow other areas to be executed with correct data, even though an earlier part of the program may not be working properly.

The Preprocessor in Debugging

By using conditional preprocessor directives, you can arrange for blocks of code to be included in your program to assist in testing. In spite of the power of the debug facilities included with many C development systems, the addition of tracing code of your own can still be useful. You have complete control

of the formatting of data to be displayed for debugging purposes, and you can even arrange for the kind of output to vary according to conditions or relationships within the program.

TRY IT OUT: USING PREPROCESSOR DIRECTIVES

I can illustrate how you can use preprocessor directives to control execution and switch debugging output on and off through a program that calls functions at random through an array of function pointers:

```
/* Program 13.1 Debugging using preprocessing directives */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Macro to generate pseudo-random number from 0 to NumValues */
#define random(NumValues) ((int)(((double)(rand())*(NumValues))/(RAND_MAX+1.0)))

#define iterations 6
#define test                            /* Select testing output       */
#define testf                           /* Select function call trace  */
#define repeatable                      /* Select repeatable execution */

/* Function prototypes */
int sum(int, int);
int product(int, int);
int difference(int, int);

int main(void)
{
  int funsel = 0;                     /* Index for function selection       */
  int a = 10, b = 5;                  /* Starting values                    */
  int result = 0;                     /* Storage for results                */

  /* Function pointer array declaration */
  int (*pfun[])(int, int) = {sum, product, difference};

  /* Conditional code for repeatable execution */
  #ifdef repeatable
  srand(1);
  #else
  srand((unsigned int)time(NULL));    /* Seed random number generation      */
  #endif

  /* Execute random function selections */
  int element_count = sizeof(pfun)/sizeof(pfun[0]);
  for(int i = 0 ; i < iterations ; i++)
  {
    /* Generate random index to pfun array */
```

```
    funsel = random(element_count);
    if( funsel>element_count-1 )
    {
      printf("\nInvalid array index = %d", funsel);
      exit(1);
  }

   #ifdef test
    printf("\nRandom index = %d", funsel);
   #endif

    result = pfun[funsel](a , b);      /* Call random function               */
    printf("\nresult = %d", result );
  }
  return 0;
}

/* Definition of the function sum */
int sum(int x, int y)
{
#ifdef testf
  printf("\nFunction sum called args %d and %d.", x, y);
#endif

  return x + y;
}

/* Definition of the function product */
int product( int x, int y )
{
  #ifdef testf
  printf("\nFunction product called args %d and %d.", x, y);
  #endif

  return x * y;
}

/* Definition of the function difference */
int difference(int x, int y)
{
  #ifdef testf
  printf("\nFunction difference called args %d and %d.", x, y);
  #endif

  return x - y;
}
```

How It Works

You have a macro defined at the beginning of the program:

```
#define random(NumValues) ((int)(((double)(rand())*(NumValues))/(RAND_MAX+1.0)))
```

This defines the macro `random()` in terms of the function `rand()` that's declared in `stdlib.h`. The function `rand()` generates random numbers in the range 0 to `RAND_MAX`, which is a constant defined in `<stdlib.h>`. The macro maps values from this range to produce values from 0 to `NumValues-1`. You cast the value from `rand()` to `double` to ensure that computation will be carried out as type `double`, and you cast the result overall back to `int` because that's what you want in the program. It's quite possible that your version of `<stdlib.h>` may already contain a macro for `random()` that does essentially the same thing. If so, you'll get an error message as the compiler won't allow two different definitions of the same macro. In this case, just delete the definition from the program.

I defined `random()` as a macro to show how you do it, but it would be better defined as a function because this would eliminate any potential problems that might arise with argument values to the macro.

You then have four directives that define symbols:

```
#define iterations 6
#define test                          /* Select testing output       */
#define testf                         /* Select function call trace  */
#define repeatable                    /* Select repeatable execution */
```

The first of these defines a symbol that specifies the number of iterations in the loop that executes one of three functions at random. The last three are all symbols that control the selection of code to be included in the program. Defining the `test` symbol causes code to be included that will output the value of the index that selects a function. Defining `testf` causes code that traces function calls to be included in the function definitions. When the `repeatable` symbol is defined, the `srand()` function is called with a fixed seed value, so the `rand()` function will always generate the same pseudo-random sequence and the same output will be produced on successive runs of the program. Having repeatable output during test runs of the program obviously makes the testing process somewhat easier. If you remove the directive that defines the `repeatable` symbol, `srand()` will be called with the current time value as the argument, so the seed will be different each time the program executes, and you will get different output on each execution of the program.

After setting up the initial variables used in `main()` you have the following statement declaring and initializing the `pfun` array:

```
int (*pfun[])(int, int) = {sum, product, difference};
```

This declares an array of pointers to functions with two parameters of type `int` and a return value of type `int`. The array is initialized using the names of the three functions so the array will contain three elements.

Next you have a directive that includes one of two alternative statements depending on whether the `repeatable` symbol is defined:

```
#ifdef repeatable
  srand(1);
#else
  srand((unsigned int)time(NULL));     /* Seed random number generation      */
#endif
```

If `repeatable` is defined, the statement that calls `srand()` with the argument value 1 will be included in the source for compilation. This will result in the same output each time you execute the program. Otherwise the statement with the result of the `time()` function as the argument will be included and you will get different output each time you run the program.

Look at the loop in `main()` that follows. The number of iterations is determined by the value of the iterations symbol; in this case it corresponds to 6. The first action in the loop is the following:

```
    funsel = random(element_count);
    if( funsel>element_count-1 )
    {
      printf("\nInvalid array index = %d", funsel);
      exit(1);
  }
```

This uses the `random()` macro with `element_count` as the argument. This is the number of elements in the `pfun` array and is calculated immediately prior to the loop. The preprocessor will substitute `element_count` in the macro expansion before the code is compiled. For safety, there is a check that we do indeed get a valid index value for the `pfun` array.

The next three lines are the following:

```
   #ifdef test
    printf("\nRandom index = %d", funsel);
   #endif
```

These include the `printf()` statement in the code when the `text` symbol is defined. If you remove the directive that defines `text`, the `printf()` will not be included in the program that is compiled.

The last two statements in the loop call a function through one of the pointers in the `pfun` array and output the result of the call:

```
    result = pfun[funsel](a , b);      /* Call random function              */
    printf("\nresult = %d", result );
```

Let's look at one of the functions that may be called, `product()` for example:

```
int product( int x, int y )
{
  #ifdef testf
  printf("\nFunction product called args %d and %d.", x, y);
  #endif

  return x * y;
}
```

The function definition includes an output statement if the `testf` symbol is defined. You can therefore control whether the statements in the `#ifdef` block are included here independently from the output block in `main()` that is controlled by `test`. With the program as written with both `test` and `testf` defined, you'll get trace output for the random index values generated and a message from each function as it's called, so you can follow the sequence of calls in the program exactly.

You can have as many different symbolic constants defined as you wish. As you've seen previously in this chapter, you can combine them into logical expressions using the `#if defined` form of the conditional directive.

Using the assert() Macro

The `assert()` macro is defined in the standard library header file `<assert.h>`. This macro enables you to insert tests of arbitrary expressions in your program that will cause the program to be terminated with a diagnostic message if a specified expression is false (that is, evaluates to 0). The argument to the `assert()` macro is an expression that results in an integer value, for example:

```
assert(a == b);
```

The expression will be true (nonzero) if a is equal to b. If a and b are unequal, the argument to the macro will be false and the program will be terminated with a message relating to the assertion. Termination is achieved by calling abort(), so it's an abnormal end to the program. When abort() is called, the program terminates immediately. Whether stream output buffers are flushed, open streams are closed, or temporary files are removed, it is implementation-dependent, so consult your compiler documentation on this.

In program 13.1 I could have used an assertion to verify that funsel is valid:

```
assert(funsel<element_count);
```

If funsel is not less than element_count, the expression will be false, so the program will assert. Typical output from the assertion looks like this:

```
Assertion failed:  funsel<element_count d:\examples\program13_01.c 44
```

The assertion facility allowing assertions to occur can be switched off by defining the symbol NDEBUG before the #include directive for assert.h, like this:

```
#define NDEBUG                          /* Switch off assertions */
#include <assert.h>
```

This will cause all assertions to be ignored.

With some systems, assertions are disabled by default, in which case you can enable them by undefining NDEBUG:

```
#undef NDEBUG                           /* Switch on assertions */
#include <assert.h>
```

By including the directive to undefine NDEBUG, you ensure that assertions are enabled for your source file. The #undef directive must appear before the #include directive for assert.h to be effective.

I can demonstrate this in operation with a simple example.

TRY IT OUT: DEMONSTRATING THE ASSERT() MACRO

Here's the code for a program that uses the assert() macro:

```
/* Program 13.2 Demonstrating assertions */
#undef NDEBUG                           /* Switch on assertions */
#include <stdio.h>
#include <assert.h>

int main(void)
{
  int y = 5;
  for(int x = 0 ; x < 20 ; x++)
  {
    printf("\nx = %d   y = %d", x, y);
    assert(x<y);
  }
  return 0;
}
```

Compiling and executing this with my compiler produces the following output:

```
x = 0   y = 5
x = 1   y = 5
x = 2   y = 5
x = 3   y = 5
x = 4   y = 5
x = 5   y = 5
Assertion failed: x<y , file Prog13_02.C, line 12
```

How It Works

At this point, apart from the `assert()` statement, the program shouldn't need much explanation, as it simply displays the values of `x` and `y` in the `for` loop.

The program is terminated by the `assert()` macro as soon as the condition `x < y` becomes false. As you can see from the output, this is when `x` reaches the value 5. The macro displays the output on `stderr`, which is always the display screen. Not only do you get the condition that failed displayed, but you also get the file name and line number in the file where the failure occurred. This is particularly useful with multifile programs in which the source of the error is pinpointed exactly.

Assertions are often used for critical conditions in a program in which, if certain conditions aren't met, disaster will surely ensue. You would want to be sure that the program wouldn't continue if such errors arise.

You could switch off the assertion mechanism in the example by replacing the `#undef` directive with `#define NDEBUG`. This must be placed before the `#include` statement for `<assert.h>` to be effective, although assertions may be disabled by default. With this `#define` at the beginning of Program 13.2, you'll see that you get output for all the values of `x` from 0 to 19, and no diagnostic message.

Additional Library Functions

The library functions are basic to the power of the C language. Although I've covered quite a range of standard library functions so far, it's beyond the scope of this book to discuss all the standard libraries. However, I can introduce a few more of the most commonly used functions you haven't dealt with in detail up to now.

The Date and Time Function Library

Because time is an important parameter to measure, C includes a standard library of functions called `<time.h>` that deal with time and dates. They provide output in various forms from the hardware timer in your PC.

The simplest function has the following prototype:

```
clock_t clock(void);
```

This function returns the processor time (not the elapsed time) used by the program since execution began, as a value of type `clock_t`. Your computer will typically be executing multiple processes at any given moment. The processor time is the total time the processor has been executing on behalf of the process that called the `clock()` function. The type `clock_t` is defined in `<time.h>` and is equivalent to type `size_t`. The value that is returned by the `clock()` function is measured in **clock ticks**, and to convert this value to seconds, you divide it by the value that is produced by the macro `CLOCKS_PER_SEC`, which is also defined in the library `<time.h>`. The value that results from executing `CLOCKS_PER_SEC` is

the number of clock ticks in one second and is of type clock_t. The clock() function returns -1 if an error occurs.

To determine the processor time used in executing a process, you need to record the time when the process starts executing and subtract this from the time returned when the process finishes. For example

```
clock_t start, end;
double cpu_time;
start = clock();

/* Execute the process for which you want the processor time */

end = clock();
cpu_time = ((double)(end-start)/CLOCKS_PER_SEC;
```

This fragment stores the total processor time used in cpu_time. The cast to type double is necessary in the last statement to get the correct result.

As you've seen, the time() function returns the calendar time as a value of type time_t. The calendar time is the current time in seconds since a fixed time and date. The fixed time and date is often 00:00:00GMT on January 1, 1970, for example, and this is typical of how time values are defined.

The prototype of the time() function is the following:

```
time_t time(time_t *timer);
```

If the argument isn't NULL, the current calendar time is also stored in the location pointed to by the argument. The type time_t is defined in the header file and is equivalent to long.

To calculate the elapsed time in seconds between two successive time_t values returned by time(), you can use the function difftime(), which has this prototype:

```
double difftime(time_t T2, time_t T1);
```

The function will return the value of T2 - T1 expressed in seconds as a value of type double. This value is the time elapsed between the two time() function calls that produce the time_t values, T1 and T2.

TRY IT OUT: USING TIME FUNCTIONS

You could define a function to log the elapsed time and processor time used between successive calls by using functions from <time.h> as follows:

```
/* Program 13.3 Test our timer function */
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <ctype.h>

int main(void)
{
  time_t calendar_start = time(NULL);       /* Initial calendar time    */
  clock_t cpu_start = clock();              /* Initial processor time   */
  int count = 0;                            /* Count of number of loops */
  const int iterations = 1000000;           /* Loop iterations          */
  char answer = 'y';
```

```
  printf("Initial clock time = %lu Initial calendar time = %lu\n",
                                    cpu_start, calendar_start);

  while(tolower(answer) == 'y')
  {
    for(int i = 0 ; i<iterations ; i++)
    {
      double x = sqrt(3.14159265);
    }
    printf("\n%ld square roots completed.", iterations*(++count));
    printf("\nDo you want to run some more(y or n)? ");

    scanf("\n%c", &answer);
  }

  clock_t cpu_end = clock();                /* Final cpu time           */
  time_t calendar_end = time(NULL);         /* Final calendar time      */

  printf("\nFinal clock time = %lu Final calendar time = %lu\n",
                                                  cpu_end, calendar_end);
  printf("\nCPU time for %ld iterations is %.2lf seconds\n",
           count*iterations, ((double)(cpu_end-cpu_start))/CLOCKS_PER_SEC );

  printf("\nElapsed calendar time to execute the program is %8.2lf\n",
                                  difftime(calendar_end, calendar_start));
  return 0;
}
```

On my machine I get the following output:

```
Initial clock time = 0 Initial calendar time = 1155841882

1000000 square roots completed.
Do you want to run some more(y or n)? y

2000000 square roots completed.
Do you want to run some more(y or n)? n

Final clock time = 7671 Final calendar time = 1155841890

CPU time for 2000000 iterations is 7.67 seconds

Elapsed calendar time to execute the program is     8.00
```

How It Works

This program illustrates the use of the functions `clock()`, `time()`, and `difftime()`. The `time()` function returns the current time in seconds, so you won't get values less than 1 second. Depending on the speed of your machine, you may want to adjust the number of iterations in the loop to reduce or increase the time required to execute this program. Note that the `clock()` function may not be a very accurate way of determining the processor time used in the program. You also need to keep in mind that measuring elapsed time using the `time()` function can be a second out.

You record and display the initial values for the processor time and the calendar time and set up the controls for the loop that follows with these statements:

```
time_t calendar_start = time(NULL);       /* Initial calendar time    */
clock_t cpu_start = clock();              /* Initial processor time   */
int count = 0;                            /* Count of number of loops */
const int iterations = 1000000;           /* Loop iterations          */
char answer = 'y';
printf("Initial clock time = %lu Initial calendar time = %lu\n",
                                  cpu_start, calendar_start);
```

With my C library, the types `clock_t` and `time_t` are defined as type `unsigned long` in the `time.h` header file. You should check how the types are defined in your library and adjust the format specifications for these values if necessary.

You then have a loop controlled by the character stored in `answer`, so the loop will execute as long as you want it to continue:

```
while(tolower(answer) == 'y')
{
  for(int i = 0 ; i<iterations ; i++)
  {
    double x = sqrt(3.14159265);
  }

   printf("\n%ld square roots completed.", iterations*(++count));
   printf("\nDo you want to run some more(y or n)? ");

  scanf("\n%c", &answer);
}
```

The inner loop calls the `sqrt()` function that is declared in the `<math.h>` header `iterations` times, so this is just to occupy some processor time. If you are leisurely in your entry of a response to the prompt for input, this should extend the elapsed time. Note the newline escape sequence in the beginning of the first argument to `scanf()`. If you leave this out, your program will loop indefinitely, because `scanf()` will not ignore whitespace characters in the input stream buffer.

Finally, you output the final values returned by `clock()` and `time()`, and calculate the processor and calendar time intervals:

```
clock_t cpu_end = clock();                /* Final cpu time           */
time_t calendar_end = time(NULL);         /* Final calendar time      */

printf("\nFinal clock time = %lu Final calendar time = %lu\n",
                                                cpu_end, calendar_end);
printf("\nCPU time for %ld iterations is %.2lf seconds\n",
         count*iterations, ((double)(cpu_end-cpu_start))/CLOCKS_PER_SEC );

printf("\nElapsed calendar time to execute the program is %8.2lf\n",
                                  difftime(calendar_end, calendar_start));
```

The library that comes with your C compiler may well have additional nonstandard functions for obtaining processor time that are more reliable than the `clock()` function.

■ **Caution** Note that the processor clock can wrap around, and the resolution with which processor time is measured can vary between different hardware platforms. For example, if the processor clock is a 32-bit value that has a microsecond resolution, the clock will wrap back to zero roughly every 72 minutes.

Getting the Date

Having the time in seconds since a date over a quarter of a century ago is interesting, but it's often more convenient to get today's date as a string. You can do this with the function `ctime()`, which has this prototype:

```
char *ctime(const time_t *timer);
```

The function accepts a pointer to a `time_t` variable as an argument that contains a calendar time value returned by the `time()` function. It returns a pointer to a 26-character string containing the day, the date, the time, and the year, which is terminated by a newline and a `'\0'`.

A typical string returned might be the following:

```
"Mon Aug 25  10:45:56 2003\n\0"
```

You might use the `ctime()` function like this:

```
time_t calendar = 0;
calendar = time(NULL);                  /* Store calendar time                 */
  printf("\n%s", ctime(&calendar));     /* Output calendar time as date string */
```

You can also get at the various components of the time and date from a calendar time value by using the library function `localtime()`. This function has the following prototype:

```
struct tm *localtime(const time_t *timer);
```

This function accepts a pointer to a `time_t` value and returns a pointer to a structure type `tm`, which is defined in the `<time.h>` header file. The structure contains the members listed in Table 13-1.

Table 13-1. *Members of the `tm` Structure*

Member	Description
`tm_sec`	Seconds after the minute on 24-hour clock
`tm_min`	Minutes after the hour on 24-hour clock (0 to 59)
`tm_hour`	The hour on 24-hour clock (0 to 23)
`tm_mday`	Day of the month (1 to 31)
`tm_mon`	Month (0 to 11)
`tm_year`	Year (current year minus 1900)
`tm_wday`	Weekday (Sunday is 0; Saturday is 6)
`tm_yday`	Day of year (0 to 365)
`tm_isdst`	Daylight saving flag. Positive for daylight saving time 0 for not daylight saving time –1 for not known

All these structure members are of type `int`. The `localtime()` function returns a pointer to the same structure each time you call it and the structure members are overwritten on each call. If you want to keep any of the member values, you need to copy them elsewhere before the next call to `localtime()`, or you could create your own `tm` structure and save the whole lot if you really need to.

The time that the `localtime()` function produces is local to where you are. If you want to get the time in a `tm` structure that reflects UTC (Coordinated Universal Time) you can use the `gmtime()` function. This also expects an argument of type `time_t` and returns a pointer to a `tm` structure.

Here's a code fragment that will output the day and the date from the members of the `tm` structure:

```
time_t calendar = 0;                /* Holds calendar time        */
struct tm *time_data;               /* Holds address of tm struct */
const char *days[] = {"Sunday",   "Monday", "Tuesday", "Wednesday",
                      "Thursday", "Friday", "Saturday"              };
constchar *months[] = {"January", "February", "March",
                       "April",    "May",     "June",
                       "July",    "August",   "September",
                       "October", "November", "December"  };

calendar = time(NULL);              /* Get current calendar time */
printf("\n%s", ctime(&calendar));
time_data = localtime(&calendar);
printf("Today is %s %s %d %d\n",
                    days[time_data->tm_wday], months[time_data->tm_mon],
                    time_data->tm_mday,       time_data->tm_year+1900);
```

You've defined arrays of strings to hold the days of the week and the months. You use the appropriate member of the structure that has been set up by the call to the `localtime()` function. You use the day in the month and the year values from the structure directly. You can easily extend this to output the time.

TRY IT OUT: GETTING THE DATE

It's very easy to pick out the members you want from the structure of type `tm` returned from the function `localtime()`. You can demonstrate this with the following example:

```
/* Program 13.4        Getting date data with ease */
#include <stdio.h>
#include <time.h>
int main(void)
{
  const char *Day[7] = {
                  "Sunday"  , "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"
                      };
  const char *Month[12] = {
                    "January",   "February", "March",    "April",
                    "May",       "June",     "July",     "August",
                    "September", "October",  "November", "December"
                       };
  const char *Suffix[4] = { "st", "nd", "rd", "th" };
  enum sufindex { st, nd, rd, th } sufsel = th;       /* Suffix selector */
```

```
  struct tm *OurT = NULL;              /* Pointer for the time structure */
  time_t Tval = 0;                     /* Calendar time                  */

  Tval = time(NULL);                   /* Get calendar time              */
  OurT = localtime(&Tval);             /* Generate time structure        */

  switch(OurT->tm_mday)
  {
    case 1: case 21: case 31:
      sufsel= st;
      break;
    case 2: case 22:
      sufsel= nd;
      break;
    case 3: case 23:
      sufsel= rd;
      break;
    default:
      sufsel= th;
      break;
  }

  printf("Today is %s the %d%s %s %d", Day[OurT->tm_wday],
    OurT->tm_mday, Suffix[sufsel], Month[OurT->tm_mon], 1900 + OurT->tm_year);
  printf("\nThe time is %d : %d : %d",
    OurT->tm_hour, OurT->tm_min, OurT->tm_sec );
  return 0;
}
```

Here's an example of output from this program:

```
Today is Friday the 18th August 2006
The time is 11 : 49 : 53
```

How It Works

In this example, the first declarations in `main()` are as follows:

```
  const char *Day[7] = {
                  "Sunday"  , "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"
                      };
  const char *Month[12] = {
                    "January",   "February", "March",    "April",
                    "May",       "June",     "July",     "August",
                    "September", "October",  "November", "December"
                        };
  const char *Suffix[] = { "st", "nd", "rd", "th" };
```

Each defines an array of pointers to `char`. The first holds the days of the week, the second contains the months in the year, and the third holds the suffixes to numerical values for the day in the month when representing dates. You could leave out the array dimensions in the first two declarations and the compiler would compute them for you, but in this case you're reasonably confident about both these numbers, so this is an instance in which putting them

in helps to avoid an error. The const qualifier specifies that the strings pointed to are constants and should not be altered in the code.

The enumeration provides a mechanism for selecting an element from the Suffix array:

```
enum sufindex { st, nd, rd, th } sufsel = th;       /* Suffix selector */
```

The enumeration constants, st, nd, rd, and th will be assigned values 0 to 3 by default, so we can use the sufsel variable as an index to access elements in the Suffix array.

You also declare a structure variable in the following declaration:

```
struct tm *OurT = NULL;              /* Pointer for the time structure */
```

This provides space to store the pointer to the structure returned by the function local_time().

You first obtain the current time in Tval using the function time(). You then use this value to generate the values of the members of the structure returned by the function localtime(). If you want to keep the data in the structure, you need to copy it before calling the localtime() function again, as it would be overwritten. Once you have the structure from localtime(), you execute the switch:

```
switch( OurT->tm_mday )
{
    case 1: case 21: case 31:
      sufsel= st;
      break;
    case 2: case 22:
      sufsel= nd;
      break;
    case 3: case 23:
      sufsel= rd;
      break;
    default:
      sufsel= th;
      break;
}
```

The sole purpose of this is to select what to append to the date value. Based on the member tm_mday, the switch selects an index to the array Suffix[] for use when outputting the date by setting the sufsel variable to the appropriate enumeration constant value.

The day, the date, and the time are displayed, with the day and month strings obtained by indexing the appropriate array with the corresponding structure member value. The addition of 1900 to the value of the member tm_year is because this value is measured relative to the year 1900.

You can also use the mktime() function to determine the day of the week for a given date. The function has the following prototype:

```
time_t mktime(struct tm *ptime);
```

You can pass a pointer to a tm structure to a function with the tm_mon, tm_day, and tm_year values set to a date you are interested in. The values of the tm_wkday and tm_yday members of the structure will be ignored, and if the operation is successful, the values will be replaced with the values that are correct for the date you have supplied. The function returns the calendar time as a value of type time_t if the operation is successful, or –1 if the date cannot be represented as a time_t value, causing the operation to fail. Let's see it working in an example.

TRY IT OUT: GETTING THE DAY FOR A DATE

It's very easy to pick out the members you want from the structure of type `tm` returned from the function `localtime()`. You can demonstrate this with the following example:

```
/* Program 13.5        Getting the day for a given date */
#include <stdio.h>
#include <time.h>
int main(void)
{
  const char *Day[7] = {
                  "Sunday"  , "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"
                      };
  const char *Month[12] = {
                    "January",   "February", "March",    "April",
                    "May",       "June",     "July",     "August",
                    "September", "October",  "November", "December"
                        };
  const char *Suffix[4] = { "st", "nd", "rd", "th" };
  enum sufindex { st, nd, rd, th } sufsel = th;  /* Suffix selector */

  int day = 0;                          /* Stores a day...           */
  int month = 0;                        /* month...                  */
  int year = 0;                         /* and year for a date       */

  struct tm birthday;                   /* A birthday time structure */

  /* Set the structure members we don't care about */
  birthday.tm_hour = birthday.tm_min = 0;
  birthday.tm_sec = 1;
  birthday.tm_isdst = -1;

  printf("Enter your birthday as integers, day month year."
            "\ne.g. Enter 1st February 1985 as 1 2 1985. : ");
  scanf(" %d %d %d", &day, &month, &year);

  birthday.tm_mon = month-1;
  birthday.tm_mday = day;
  birthday.tm_year = year-1900;

  if(mktime(&birthday) == (time_t)-1)
  {
    printf("\nOperation failed.");
    return 0;
  }

  switch(birthday.tm_mday)
  {
      sufsel= st;
      break;
    case 2: case 22:
      sufsel= nd;
      break;
```

```
 case 3: case 23:
      sufsel= rd;
      break;
    default:
      sufsel= th;
      break;
  }

  printf("\nYour birthday, the %d%s %s %d, was a %s",
                birthday.tm_mday, Suffix[sufsel], Month[birthday.tm_mon],
                           1900 + birthday.tm_year, Day[birthday.tm_wday]);

  return 0;
}
```

Here's a sample of output from this example:

```
Enter your birthday as integers, day month year.
e.g. Enter 1st February 1985 as 1 2 1985. : 15 6 1985

Your birthday, the 15th June 1985, was a Saturday
```

How It Works

You create arrays of constant strings for the day month and date suffixes as you did in Program 13.4. You then create a `tm` structure and initialize some of its members:

```
struct tm birthday;                  /* A birthday time structure */

/* Set the structure members we don't care about */
birthday.tm_hour = birthday.tm_min = 0;
birthday.tm_sec = 1;
birthday.tm_isdst = -1;
```

You set the value of the `tm_isdst` member to –1 because you don't know whether it applies for the date that will be entered.

After outputting a prompt, you read values for the day, month, and year of a birthday date from the keyboard and set the values entered in the birthday structure:

```
printf("\nEnter your birthday as integers, day month year."
          "\ne.g. Enter 1st February 1985 as 1 2 1985. : ");
scanf(" %d %d %d", &day, &month, &year);

birthday.tm_mon = month-1;
birthday.tm_mday = day;
birthday.tm_year = year-1900;
```

With the date set, you can get the `tm_wday` and `tm_yday` members set according by calling the `mktime()` function:

```
if(mktime(&birthday) == (time_t)-1)
{
  printf("\nOperation failed.");
  return 0;
}
```

The `if` statement checks whether the function returns –1, indicating that the operation has failed. In this case you simply output a message and terminate the program.

Finally, the day corresponding to the birth date entered is displayed in the same way as in the previous example.

Summary

In this chapter I discussed the preprocessor directives that you use to manipulate and transform the code in a source file before it is compiled. Your standard library header files are an excellent source of examples of coding preprocessing directives. You can view these examples with any text editor. Virtually all of the capabilities of the preprocessor are used in the libraries, and you'll find a lot of C source code there too. It's also useful to familiarize yourself with the contents of the libraries, as you can find many things not necessarily described in the library documentation. If you aren't sure what the type `clock_t` is, for example, just look in the library header file `<time.h>`, where you'll find the definition.

The debugging capability that the preprocessor provides is useful, but you will find that the tools that are provided for debugging with many C programming systems are much more powerful. For serious program development the debugging tools are as important as the efficiency of the compiler.

If you've reached this point and are confident that you understand and can apply what you've read, you should now be comfortable with programming in C. All you need now is more practice to improve your expertise. To get better at programming, there's no alternative to practice, and the more varied the types of programs you write, the better. You can always improve your skills, but, fortunately—or unfortunately, depending on your point of view—it's unlikely that you'll ever reach perfection in programming, as it's almost impossible to sit down and write a bug-free program of any size from the outset. However, every time you get a new piece of code to work as it should, it will always generate a thrill and a feeling of satisfaction. Enjoy your programming!

Exercises

The following exercises enable you to try out what you learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 13-1. Define a macro, `COMPARE(x, y)`, that will result in the value –1 if `x < y`, 0 if `x == y`, and 1 if `x > y`. Write an example to demonstrate that your macro works as it should. Can you see any advantage that your macro has over a function that does the same thing?

Exercise 13-2. Define a function that will return a string containing the current time in 12-hour format (a.m./p.m.) if the argument is 0, and in 24-hour format if the argument is 1. Demonstrate that your function works with a suitable program.

Exercise 13-3. Define a macro, `print_value(expr)`, that will output on a new line `exp = result` where `result` is the value that results from evaluating `expr`. Demonstrate the operation of your macro with a suitable program.

APPENDIX A

Computer Arithmetic

In the chapters of this book, I've deliberately kept discussion of arithmetic to a minimum. However, it's important, so I'm going to quickly go over the subject in this appendix. If you feel confident in your math skills, this review will be old hat to you. If you find the math parts tough, this section should show you how easy it really is.

Binary Numbers

First, let's consider exactly what you intend when you write a common, everyday decimal number such as 324 or 911. Obviously what you mean is *three hundred and twenty-four* or *nine hundred and eleven*. Put more precisely, you mean the following:

324 is $3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$, which is $3 \times 10 \times 10 + 2 \times 10 + 4$

911 is $9 \times 10^2 + 1 \times 10^1 + 1 \times 10^0$, which is $9 \times 10 \times 10 + 1 \times 10 + 1$

We call this **decimal notation** because it's built around powers of 10. (This is derived from the Latin **decimalis**, meaning "of tithes," which was a tax of 10 percent. Ah, those were the days . . .)

Representing numbers in this way is very handy for people with 10 fingers and/or 10 toes, or indeed 10 of any kind of appendage. However, your PC is rather less handy, being built mainly of switches that are either on or off. It's OK for counting up to 2, but not spectacular at counting to 10. I'm sure you're aware that this is the primary reason why your computer represents numbers using base 2 rather than base 10. Representing numbers using base 2 is called the **binary** system of counting. With numbers expressed using base 10, digits can be from 0 to 9 inclusive, whereas with binary numbers digits can only be 0 or 1, which is ideal when you only have on/off switches to represent them. In an exact analogy to the system of counting in base 10, the binary number 1101, for example, breaks down like this:

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

which is

$1 \times 2 \times 2 \times 2 + 1 \times 2 \times 2 + 0 \times 2 + 1$

This amounts to 13 in the decimal system. In Table A-1 you can see the decimal equivalents of all the possible numbers you can represent using eight binary digits (a binary digit is more commonly known as a **bit**).

Notice that using the first seven bits you can represent numbers from 0 to 127, which is a total of 2^7 numbers, and that using all eight bits you get 256 or 2^8 numbers. In general, if you have n bits, you can represent 2^n integers, with values from 0 to $2^n - 1$.

Table A-1. *Decimal Equivalents of 8-Bit Binary Values*

Binary	Decimal	Binary	Decimal
0000 0000	0	1000 0000	128
0000 0001	1	1000 0001	129
0000 0010	2	1000 0010	130
. . .	. . .	. . .	. . .
0001 0000	16	1001 0000	144
0001 0001	17	1001 0001	145
. . .	. . .	. . .	. . .
0111 1100	124	1111 1100	252
0111 1101	125	1111 1101	253
0111 1110	126	1111 1110	254
0111 1111	127	1111 1111	255

Adding binary numbers inside your computer is a piece of cake, because the "carry" from adding corresponding digits can only be 0 or 1, and very simple circuitry can handle the process. Figure A-1 shows how the addition of two 8-bit binary values would work.

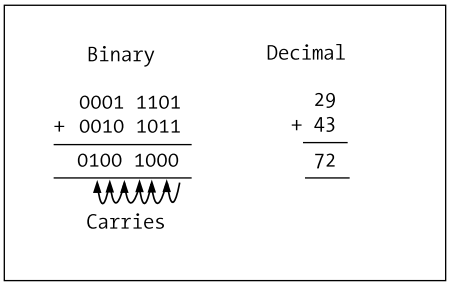


Figure A-1. *Adding binary values*

Hexadecimal Numbers

When you start dealing with larger binary numbers, a small problem arises. Look at this one:

1111 0101 1011 1001 1110 0001

Binary notation here starts to be more than a little cumbersome for practical use, particularly when you consider that if you work out what this is in decimal, it's only 16,103,905—a miserable eight decimal digits. You can sit more angels on the head of a pin than that. Clearly we need a more economical way of writing this, but decimal isn't always appropriate. Sometimes (as you saw in Chapter 3) you might need to be able to specify that the 10th and 24th bits from the right are set to 1, but without the overhead of writing out all the bits in binary notation. To figure out the decimal integer required to do this sort of thing is hard work, and there's a good chance you'll get it wrong

anyway. A much easier solution is to use hexadecimal notation in which the numbers are represented using base 16.

Arithmetic to base 16 is a much more convenient option, and it fits rather well with binary. Each hexadecimal digit can have values from 0 to 15 (the digits from 10 to 15 being represented by letters A to F, as shown in Table A-2), and values from 0 to 15 correspond nicely with the range of values that four binary digits can represent.

Table A-2. *Hexadecimal Digits and Their Values in Decimal and Binary*

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Because a hexadecimal digit corresponds to four binary digits, you can represent a large binary number as a hexadecimal number simply by taking groups of four binary digits starting from the right and writing the equivalent hexadecimal digit for each group. Look at the following binary number:

1111 0101 1011 1001 1110 0001

Taking each group of four bits in turn and replacing it with the corresponding hexadecimal digit from the table, this number expressed in hexadecimal notation will come out as follows:

F 5 B 9 E 1

You have six hexadecimal digits corresponding to the six groups of four binary digits. Just to prove that it all works out with no cheating, you can convert this number directly from hexadecimal to decimal by again using the analogy with the meaning of a decimal number. The value of this hexadecimal number therefore works out as follows:

F5B9E1 as a decimal value is given by

$15 \times 16^5 + 5 \times 16^4 + 11 \times 16^3 + 9 \times 16^2 + 14 \times 16^1 + 1 \times 16^0$

This turns out to be

15,728,640 + 327,680 + 45,056 + 2,304 + 224 + 1

Thankfully, this adds up to the same number you get when converting the equivalent binary number to a decimal value: 16,103,905.

Negative Binary Numbers

There's another aspect to binary arithmetic that you need to understand: negative numbers. So far, you've assumed that everything is positive—the optimist's view, if you will—and so the glass is still half full. But you can't avoid the negative side of life—the pessimist's perspective—that the glass is already half empty. How can a negative number be represented inside a computer? Well, you have only binary digits at your disposal, so the solution has to be to use one of those to indicate whether the number is negative or positive.

For numbers that you want to allow to have negative values (referred to as **signed numbers**), you must first decide on a fixed length (in other words, the number of binary digits) and then designate the leftmost binary digit as a **sign bit**. You have to fix the length to avoid any confusion about which bit is the sign bit.

Because your computer's memory consists of 8-bit bytes, the binary numbers are going to be stored in some multiple (usually a power of 2) of 8 bits. Thus, you can have some numbers with 8 bits, some with 16 bits, and some with 32 bits (or whatever), and as long as you know what the length is in each case, you can find the sign bit—it's just the leftmost bit. If the sign bit is 0, the number is positive, and if it's 1, the number is negative.

This seems to solve the problem, and in some computers it does. Each number consists of a sign bit that is 0 for positive values and 1 for negative values, plus a given number of bits that specify the absolute value of the number—unsigned, in other words. Changing +6 to –6 just involves flipping the sign bit from 0 to 1. Unfortunately this representation carries a lot of overhead with it in terms of the complexity of the circuits that are needed to perform arithmetic with this number representation. For this reason, most computers take a different approach.

Ideally when two integers are added, you don't want the computer to be messing about, checking whether either or both of the numbers are negative. You just want to use simple **add** circuitry, regardless of the signs of the operands. The add operation will combine corresponding binary digits to produce the appropriate bit as a result, with a carry to the next digit where necessary. If you add –8 in binary to +12, you would really like to get the answer +4, using the same circuitry that would apply if you were adding +3 and +8.

If you try this with your simplistic solution, which is just to set the sign bit of the positive value to 1 to make it negative, and then perform the arithmetic with conventional carries, it doesn't quite work:

12 in binary is 0000 1100.

–8 in binary (you suppose) is 1000 1000.

If you now add these together, you get 1001 0100.

This seems to be –20, which isn't what you wanted at all. It's definitely not +4, which you know is 0000 0100. "Ah," I hear you say, "you can't treat a sign just like another digit." But that is just what you *do* want to do.

Let's see how the computer would like you to represent –8 by trying to subtract +12 from +4, as that should give you the right answer:

+4 in binary is 0000 0100.

+12 in binary is 0000 1100.

Subtract the latter from the former and you get 1111 1000.

For each digit from the fourth from the right onward, you had to "borrow" 1 to do the subtraction, just as you would when performing ordinary decimal arithmetic. This result is supposed to be –8, and even though it doesn't look like it, that's exactly what it is. Just try adding it to +12 or +15 in binary, and you'll see that it works.

Of course, if you want to produce –8 you can always do so by subtracting +8 from 0.

What *exactly* did you get when you subtracted 12 from 4 or +8 from 0, for that matter? It turns out that what you have here is called the **2's complement** representation of a negative binary number, and you can produce this from any positive binary number by a simple procedure that you can perform in your head. At this point, I need to ask a little faith on your part and avoid getting into explanations of *why* it works. I'll just show you how the 2's complement form of a negative number can be constructed from a positive value, and you can prove to yourself that it does work. Let's return to the previous example, in which you need the 2's complement representation of –8.

You start with +8 in binary:

0000 1000

You now "flip" each binary digit, changing 0s to 1s and vice versa:

1111 0111

This is called the **1's complement** form, and if you now add 1 to this, you'll get the 2's complement form:

1111 1000

This is exactly the same as the representation of –8 you get by subtracting +12 from +4. Just to make absolutely sure, let's try the original sum of adding –8 to +12:

+12 in binary is 0000 1100.

Your version of –8 is 1111 1000.

If you add *these* together, you get 0000 0100.

The answer is 4—magic. It works! The "carry" propagates through all the leftmost 1s, setting them back to 0. One fell off the end, but you shouldn't worry about that—it's probably compensating for the one you borrowed from the end in the subtraction operation that produced the –8 in the first place. In fact, what's happening is that you're making an assumption that the sign bit, 1 or 0, repeats forever to the left. Try a few examples of your own; you'll find it always works automatically. The really great thing about using 2's complement representation of negative numbers is that it makes arithmetic very easy (and fast) for your computer.

Big-Endian and Little-Endian Systems

As I've discussed, integers generally are stored in memory as binary values in a contiguous sequence of bytes, commonly groups of 2, 4, or 8 bytes. The question of the sequence in which the bytes appear can be very important—it's one of those things that doesn't matter until it matters, and then it *really* matters.

Let's consider the decimal value 262,657 stored as a 4-byte binary value. I chose this value because in binary it happens to be

0000 0000 0000 0100 0000 0010 0000 0001

Each byte has a pattern of bits that is easily distinguished from the others.

If you're using an Intel PC, the number will be stored as follows:

Byte address:	00	01	02	03
Data bits:	0000 0001	0000 0010	0000 0100	0000 0000

As you can see, the most significant eight bits of the value—the one that's all 0s—are stored in the byte with the highest address (last, in other words), and the least significant eight bits are stored in the byte with the lowest address, which is the leftmost byte. This arrangement is described as **little-endian**.

If you're using a mainframe computer, a RISC workstation, or a Mac machine based on a Motorola processor, the same data is likely to be arranged in memory as follows:

Byte address:	00	01	02	03
Data bits:	0000 0000	0000 0100	0000 0010	0000 0001

Now the bytes are in reverse sequence with the most significant eight bits stored in the leftmost byte, which is the one with the lowest address. This arrangement is described as **big-endian**.

■ **Note** Within each byte, the bits are arranged with the most significant bit on the left and the least significant bit on the right, regardless of whether the byte order is big-endian or little-endian.

This is all very interesting, you may say, but why should it matter? Most of the time it doesn't. More often than not you can happily write your C program without knowing whether the computer on which the code will execute is big-endian or little-endian. It *does* matter, however, when you're processing binary data that comes from another machine. Binary data will be written to a file or transmitted over a network as a sequence of bytes. It's up to you how you interpret it. If the source of the data is a machine with a different endian-ness from the machine on which your code is running, you must reverse the order of the bytes in each binary value. If you don't, you have garbage.

■ **Note** For those who collect curious background information, the terms **big-endian** and **little-endian** are drawn from the book *Gulliver's Travels* by Jonathan Swift. In the story, the emperor of Lilliput commands all his subjects to always crack their eggs at the smaller end. This is a consequence of the emperor's son having cut his finger following the traditional approach of cracking his egg at the big end. Ordinary law-abiding Lilliputian subjects who cracked their eggs at the smaller end were described as Little Endians. The Big Endians were a rebellious group of traditionalists in the Lilliputian kingdom who insisted on continuing to crack their eggs at the big end. Many were put to death as a result.

Floating-Point Numbers

We often have to deal with very large numbers—the number of protons in the universe, for example—which need around 79 decimal digits. Clearly there are a lot of situations in which you'll need more than the 10 decimal digits you get from a 4-byte binary number. Equally, there are a lot of very small numbers—for example, the amount of time in minutes it takes the typical car salesperson to accept your generous offer on a 1982 Ford LTD (and it's covered only 380,000 miles . . .). A mechanism for handling both these kinds of numbers is, as you may have guessed, **floating-point** numbers.

A floating-point representation of a number in decimal notation is a decimal value 0 with a fixed number of digits multiplied by a power of 10 to produce the number you want. It's easier to demonstrate than to explain, so let's look at some examples. The number 365 in normal decimal notation could be written in floating-point form as follows:

0.3650000E03

The *E* stands for **exponent** and precedes the power of 10 that the 0.3650000 (the mantissa) part is multiplied by to get the required value. That is

$0.3650000 \times 10 \times 10 \times 10$

which is clearly 365.

The mantissa in the number here has seven decimal digits. The number of digits of precision in a floating-point number will depend on how much memory it is allocated. A single precision floating-point value occupying 4 bytes will provide approximately seven decimal digits accuracy. I say *approximately* because inside your computer these numbers are in binary floating-point form, and a binary fraction with 23 bits doesn't exactly correspond to a decimal fraction with seven decimal digits.

Now let's look at a small number:

0.3650000E-04

This is evaluated as $.365 \times 10^{-4}$, which is .0000365—exactly the time in minutes required by the car salesperson to accept your cash.

Suppose you have a large number such as 2,134,311,179. How does this look as a floating-point number? Well, it looks like this:

0.2134311E10

It's not quite the same. You've lost three low-order digits and you've approximated your original value as 2,134,311,000. This is a small price to pay for being able to handle such a vast range of numbers, typically from 10^{-38} to 10^{+38} either positive or negative, as well as having an extended representation that goes from a minute 10^{-308} to a mighty 10^{+308}. They're called floating-point numbers for the fairly obvious reason that the decimal point "floats" and its position depends on the exponent value.

Aside from the fixed precision limitation in terms of accuracy, there's another aspect you may need to be conscious of. You need to take great care when adding or subtracting numbers of significantly different magnitudes. A simple example will demonstrate this kind of problem. You can first consider adding .365E–3 to .365E+7. You can write this as a decimal sum:

0.000365 + 3,650,000.0

This produces the following result:

3,650,000.000365

When converted back to floating-point with seven digits of precision, this becomes the following:

0.3650000E+7

So you might as well not have bothered. The problem lies directly with the fact that you carry only seven digits precision. The seven digits of the larger number aren't affected by any of the digits of the smaller number because they're all further to the right. Funnily enough, you must also take care when the numbers are nearly equal. If you compute the difference between such numbers, you may end up with a result that has only one or two digits precision. It's quite easy in such circumstances to end up computing with numbers that are totally garbage.

APPENDIX B

ASCII Character Code Definitions

The first 32 American Standard Code for Information Interchange (ASCII) characters provide control functions. Many of these haven't been referenced in this book but are included here for completeness. In Table B-1, only the first 128 characters are included. The remaining 128 characters include further special symbols and letters for national character sets.

Table B-1. *ASCII Character Code Values*

Decimal	Hexadecimal	Character	Control
000	00	Null	NUL
001	01	☺	SOH
002	02	●	STX
003	03	♥	ETX
004	04	♦	EOT
005	05	♣	ENQ
006	06	♠	ACK
007	07	•	BEL (audible bell)
008	08	--	Backspace
009	09	--	HT
010	0A	--	LF (linefeed)
011	0B	--	VT (vertical tab)
012	0C	--	FF (form feed)
013	0D	--	CR (carriage return)
014	0E	--	SO
015	0F	--	SI
016	10	--	DLE
017	11	--	DC1
018	12	--	DC2
019	13	--	DC3

Table B-1. *ASCII Character Code Values (Continued)*

Decimal	Hexadecimal	Character	Control
020	14	--	DC4
021	15	--	NAK
022	16	--	SYN
023	17	--	ETB
024	18	--	CAN
025	19	--	EM
026	1A	→	SUB
027	1B	←	ESC (escape)
028	1C	└	FS
029	1D	--	GS
030	1E	--	RS
031	1F	--	US
032	20	--	Space
033	21	!	--
034	22	"	--
035	23	#	--
036	24	$	--
037	25	%	--
038	26	&	--
039	27	'	--
040	28	(	--
041	29	)	--
042	2A	*	--
043	2B	+	--
044	2C	'	--
045	2D	-	--
046	2E	.	--
047	2F	/	--
048	30	0	--
049	31	1	--
050	32	2	--
051	33	3	--

Table B-1. *ASCII Character Code Values (Continued)*

Decimal	Hexadecimal	Character	Control
052	34	4	--
053	35	5	--
054	36	6	--
055	37	7	--
056	38	8	--
057	39	9	--
058	3A	:	--
059	3B	;	--
060	3C	<	--
061	3D	=	--
062	3E	>	--
063	3F	?	--
064	40	@	--
065	41	A	--
066	42	B	--
067	43	C	--
068	44	D	--
069	45	E	--
070	46	F	--
071	47	G	--
072	48	H	--
073	49	I	--
074	4A	J	--
075	4B	K	--
076	4C	L	--
077	4D	M	--
078	4E	N	--
079	4F	O	--
080	50	P	--
081	51	Q	--
082	52	R	--
083	53	S	--

Table B-1. *ASCII Character Code Values (Continued)*

Decimal	Hexadecimal	Character	Control
084	54	T	--
085	55	U	--
086	56	V	--
087	57	W	--
088	58	X	--
089	59	Y	--
090	5A	Z	--
091	5B	[	--
092	5C	\	--
093	5D	]	--
094	5E	^	--
095	5F	_	--
096	60	'	--
097	61	a	--
098	62	b	--
099	63	c	--
100	64	d	--
101	65	e	--
102	66	f	--
103	67	g	--
104	68	h	--
105	69	i	--
106	6A	j	--
107	6B	k	--
108	6C	l	--
109	6D	m	--
110	6E	n	--
111	6F	o	--
112	70	p	--
113	71	q	--
114	72	r	--
115	73	s	--

Table B-1. *ASCII Character Code Values (Continued)*

Decimal	Hexadecimal	Character	Control
116	74	t	--
117	75	u	--
118	76	v	--
119	77	w	--
120	78	x	--
121	79	y	--
122	7A	z	--
123	7B	{	--
124	7C	\|	--
125	7D	}	--
126	7E	~	--
127	7F	Delete	--

Note that the null character in the table is not necessarily the same as `NULL`, which is defined in the C library and whose value is implementation-dependent.

APPENDIX C

Reserved Words in C

The words in the following list are keywords in C, so you must not use them for other purposes, such as variable names or function names.

```
auto        for         struct
break       goto        switch
case        if          typedef
char        inline      union
const       int         unsigned
continue    long        void
default     register    volatile
do          restrict    while
double      return      _Bool
else        short       _Complex
enum        signed      _Imaginary
extern      sizeof
float       static
```

APPENDIX D

Input and Output Format Specifications

This appendix summarizes all the format specifications you have available for stream input and output. You use these with the standard streams `stdin`, `stdout`, and `stderr`, as well as text file streams.

Output Format Specifications

There are three standard library functions for formatted output: the `printf()` that writes to the standard output stream `stdout` (which by default is the command line), the `sprintf()` function that writes to a string, and the `fprintf()` function that writes to a file. These functions have the following form:

```
int printf(const char* format_string, . . .);
int sprintf(char* source_string, const char* format_string, . . .);
int fprintf(FILE* file_stream, const char* format_string, . . .);
```

The ellipsis at the end of the parameter list indicates that there can be zero or more arguments supplied. These functions return the number of bytes written, or a negative value if an error occurs. The format string can contain ordinary characters (including escape sequences) that are written to the output, together with format specifications for outputting the values of succeeding arguments.

An output format specification always begins with a `%` character and has the following general form:

```
%[flags][width][.precision][size_flag]type
```

The items between square brackets are all optional, so the only mandatory bits are the `%` character at the beginning and the `type` specifier for the type of conversion to be used.

The significance of each of the optional parts is as follows:

`[flags]` are zero or more conversion flags that control how the output is presented. The flags you can use are shown in Table D-1.

Table D-1. *Conversion Flags*

Flag	Description
`+`	Include the sign in the output, + or–. For example, `%+d` will output a decimal integer with the sign always included.
`space`	Use space or – for the sign, that is, a positive value is preceded by a space. This is useful for aligning output when there may be positive and negative values in a column of output. For example, `% d` will output a decimal integer with a space for the sign with positive values.

Table D-1. *Conversion Flags (Continued)*

Flag	Description
-	Left-justify the output in the field width with spaces padding to the right if necessary. For example, `%-10d` will output an integer as a decimal value left-justified in a field width of ten characters. The `%-+10d` specification will output a decimal integer with the sign always appearing, and left-justified in a field width of ten characters.
#	Prefix hexadecimal output values with `0x` or `0X` (corresponding to `x` and `X` conversion type specification respectively), and octal values with 0.
0	Use 0 as the pad character to the left in a right-justified output. For example, `%012d` will output a decimal integer right-justified in a field width of 12 characters, padded to the left with zeros as necessary.

`[width]` specifies the minimum field width for the output value. The width you specify will be exceeded if the value does not fit within the specified minimum width. For example, `%15u` outputs an unsigned integer value right-justified in a field width of 15 characters padded to the left with spaces as necessary.

`[.precision]` specifies the number of places following the decimal point in the output for a floating-point value. For example, `%15.6f` outputs a floating-point value in a minimum field width of 15 characters with four places after the decimal point.

`[size_flag]` is a size specification for the value that modifies the meaning of the type specification. Possible size specifications are `l` (lowercase *L*), `L`, `ll` (two lowercase *L*'s), or `h`. The size specification you can use in any given situation depends on the type specification you are using, as shown in Table D-2.

`type` is a character specifying the type of conversion to be applied to the value to be output, as shown in Table D-2.

Table D-2. *Conversion Type and Size Specifications*

Conversion Type	Description
`d`, `i`	The value is assumed to be of type `int` and the output is as a decimal integer. With the `h` size modifier (`hd` or `hi`) the argument is assumed to be type `short`. With the `l` size modifier (`ld` or `li`) the argument is assumed to be type `long`. With the `ll` modifier (`lld` or `lli`) the argument is assumed to be type `long long`.
`u`	The value is assumed to be of type `unsigned int` and the output is as an unsigned decimal integer. With the `h` size modifier (`hu`) the argument is assumed to be type `unsigned short`. With the `l` size modifier (`lu`) the argument is assumed to be type `unsigned long`. With the `ll` modifier (`llu`) the argument is assumed to be type `unsigned long long`.
`o`	The value is assumed to be of type `unsigned int` and the output is as an unsigned octal value. With the `h` size modifier (`ho`) the argument is assumed to be type `unsigned short`. With the `l` size modifier (`lo`) the argument is assumed to be type `unsigned long`. With the `ll` modifier (`llo`) the argument is assumed to be type `unsigned long long`.

Table D-2. *Conversion Type and Size Specifications*

Conversion Type	Description
x or X	The value is assumed to be of type `unsigned int` and the output is as an unsigned hexadecimal value. The hexadecimal digits `a` to `f` are used if the lowercase type conversion specification is used, and `A` to `F` otherwise. With the `h` size modifier (`ho`) the argument is assumed to be type `unsigned short`. With the `l` size modifier (`lo`) the argument is assumed to be type `unsigned long`. With the `ll` modifier (`llo`) the argument is assumed to be type `unsigned long long`.
c	The value is assumed to be of type `char` and the output is as a character. With the `l` size modifier (`lc`) the argument is assumed to be the wide character type `wchar_t`.
e or E	The value is assumed to be of type `double` and the output is as a floating-point value in scientific notation (with an exponent). The exponent value in the output will be preceded by `e` when you use the lowercase type conversion, `e`, and `E` otherwise. With the `L` modifier (`Le` or `LE`) the argument is assumed to be type `long double`.
f or F	The value is assumed to be of type `double` and the output is as a floating-point value in ordinary notation (without an exponent). With the `L` modifier (`Lf` or `LF`) the argument is assumed to be type `long double`.
g or G	The value is assumed to be of type `double` and the output is as a floating-point value in ordinary notation (without an exponent) unless the exponent value is greater than the precision (default value 6) or is less than –4, in which case the output will be in scientific notation. With the `L` modifier (`Lg` or `LG`) the argument is assumed to be type `long double`.
s	The argument is assumed to be a null-terminated string of characters of type `char` and characters are output until the null character is found or until the precision specification is reached if it is present. The optional precision specification represents the maximum number of characters that may be output.
S	When used with `printf()` the argument is assumed to be a null-terminated string of characters of type `wchar_t` and characters are output until the null character is found or until the precision specification is reached if it is present. The optional precision specification represents the maximum number of characters that may be output.
p	The argument is assumed to be a pointer, and because the output is an address it will be a hexadecimal value.
n	The argument is assumed to be a pointer of type `int*` (pointer to `int`) and the number of characters in the output so far is stored at the address pointed to by the argument. If you use the `h` modifier (`hn`) the argument is assumed to be type `short*` (pointer to `short`). If you use the `l` modifier (`ln`) the argument is assumed to be type `long*` (pointer to `long`). If you use the `ll` modifier (`lln`) the argument is assumed to be type `long long*` (pointer to `long long`).
%	No argument is expected and the output is the `%` character.

Input Format Specifications

For the `scanf()` function that reads data from the standard input stream, `stdin` (which by default is the keyboard), the `sscanf()` function that reads data from a string in memory, and the `fscanf()` function that reads data from a file, data is read from the source controlled by a format string that is passed as an argument to the function. These input functions have the following form:

```
int scanf(const char* format_string, pArg1, ...);
int sscanf(const char* destination_string, const char* format_string, ...);
int fscanf(FILE* file_stream, const char* format_string, ...);
```

Each of these functions returns a count of the number of data items read by the operation. The ellipsis at the end of the parameter list indicates that there can be zero or more arguments here. Don't forget, the arguments that follow the format string must always be pointers. It is a common error to use a variable that is not a pointer as an argument to one of these input functions.

The format string controlling how the input is processed can contain spaces, other characters, and format specifications, beginning with a `%` character, for data items.

A single whitespace character in the format string causes the function to ignore successive whitespace characters in the input. The first nonwhitespace character found will be interpreted as the first character of the next data item. When a newline character in the input follows a value that has been read, for example, when you are reading a single character from the keyboard using the `%c` format specification, any newline, tab, or space character that is entered will be read as the input character. This will be particularly apparent when you are reading a single character repeatedly, where the newline from the Enter key will be left in the buffer. If you want the function to ignore the whitespace in such situations, you can force the function to skip whitespace by including at least one whitespace character preceding the `%c` in the format string.

You can also include nonwhitespace characters in the input format string that are not part of a format specification. Any nonwhitespace character in the format string that is not part of a format specification must be matched by the same character in the input, otherwise the input operation ends.

The format specification for an item of data is of the following form:

```
%[*][width][size_flag]type
```

The items enclosed between square brackets are optional. The mandatory parts of the format specification are the `%` character marking the start of the format specification and the `type` conversion type specification at the end. The meanings of the optional parts are as follows:

`[*]` indicates that that the input data item corresponding to this format specification should be scanned but not stored. For example, `%*d` will scan an integer value and discard it.

`[width]` specifies the maximum number of characters to be scanned for this input value. If a whitespace character is found before width characters have been scanned, it is the end of the input for the current data item. For example, `%2d` reads up to two characters as an integer value. The width specification is useful for reading multiple inputs that are not separated by whitespace characters. You could read 12131415 and interpret it as the values 12, 13, 14, and 15 by using `"%2d%2d%2d%2d"` as the format string.

`[size_flag]` modifies the input type specified by the `type` part of the specification. Possible `size_flag` specifications include `h`, `l`, (lowercase *L*), `ll` (lowercase *L*'s), and `L`. Which of these you can use depends on the type specifier you are using, as described in Table D-3.

Table D-3. *Input Conversion Type Specifications and Modifiers*

Conversion Type	Description
c	Reads a single character as type char. When you use the l modifier (%lc) a single character is read as type wchar_t. You can also precede the c or lc specification with a decimal integer, m, to read m successive characters as a string not terminated by null. For example, %20c will read 20 successive characters. The corresponding argument should be a pointer to a character array with sufficient elements to accommodate the number of characters to be read.
d	Reads successive decimal digits as a value of type int. With the h modifier (%hd), successive digits are read and interpreted as type short. With the l modifier (%ld), successive digits are read and interpreted as type long. With the ll modifier (%lld), successive digits are read and interpreted as type long long.
u	Reads successive decimal digits as a value of type unsigned int. With the h modifier (%hu), successive digits are read and interpreted as type unsigned short. With the l modifier (%lu), successive digits are read and interpreted as type unsigned long. With the ll modifier (%llu), successive digits are read and interpreted as type unsigned long long.
o	Reads successive octal digits as a value of type unsigned int. With the h modifier (%ho), successive octal digits are read and interpreted as type unsigned short. With the l modifier (%lo), successive octal digits are read and interpreted as type unsigned long. With the ll modifier (%llo), successive octal digits are read and interpreted as type unsigned long long.
x or X	Reads successive hexadecimal digits as a value of type unsigned int. With the h modifier (%hx or %hX), successive hexadecimal digits are read and interpreted as type unsigned short. With the l modifier (%lx or %lX), successive hexadecimal digits are read and interpreted as type unsigned long. With the ll modifier (%llx or %llX), successive digits are read and interpreted as type unsigned long long.
s	Reads successive characters until a whitespace is reached and stores the address of the null-terminated string that results in the corresponding argument. If you use the l modifier (%ls) the characters are read and stored as a null-terminated wide character string.
n	Reads no input, but the number of characters that have been read from the input source up to this point is stored as an integer at the address specified by the corresponding argument, which should be of type int*.

Note that if you want to read a string that includes whitespace characters, you have the `%[set_of_characters]` form of specification available. With this specification, successive characters are read from the input source as long as they appear in the set you supply between the square brackets. Thus, the specification `%[ abcdefghijklmnopqrstuvwxyz]` will read any sequence of lowercase letters and spaces as a single string. A more useful variation on this is to precede the set of characters with a caret, `^`, as in `%[^set_of_characters]`, in which case the set of characters represents the characters that will be interpreted as ending the string input. For example, the specification `%[^,!]` will read a sequence of characters until either a comma or an exclamation point is found, which will end the string input.

Index

Symbols

- flag, output specifier, 395, 398, 399
-- operator
 see decrement operator (--)
- operator, 31
! (NOT) logical operator, 97–99
 operator precedence, 103
 when not to use, 173
character
 see hash (#) character
% character
 see percent (%) character
% operator, 32
& (address of) operator
 see address of (&) operator
& bitwise operator, 116
&& (AND) logical operator, 96–97
* character
 see also indirection operator (*)
 see also multiplication operator (*)
 input format specifiers, 377, 576
. operator, 411
/ divide operator, 31
?: conditional operator, 99–100
\ character
 see backslash (\) character
^ character
 bitwise operator, 116, 117
 specifying delimiters, 378
 string to include only specified characters, 386
| bitwise operator, 116, 117
|| (OR) logical operator, 97
~ bitwise operator, 116, 117
+ flag, output specifier, 395, 399, 573
+ operator, 31
++ operator
 see increment operator (++)
< (less than) operator, 82, 215
<< bitwise operator, 116, 118
<= (less than or equal to) operator, 92
= operator
 see assignment (=) operator
= = operator
 see equal to (= =) operator
> (greater than) operator, 82, 215
–> operator, 417
!= (not equal to) operator, 92
>= (greater than or equal to) operator, 92
>> bitwise operator, 116, 118

Numerics

0 flag, output specifier, 395, 574
1's complement, 116, 561
2's complement, 561
8-bit binary values, 558

A

a (\a) escape sequence, 12, 13, 17, 397
a mode, 469, 521
a+ mode, 521
ab mode, 486, 521
ab+ (a+b) mode, 521, 503, 512
abort function, 350
 using assert macro, 544
absolute values
 fabs function, 71
acos function, 72
addition operator
 += shortcut, 70
 arithmetic operators, 32
 complex number formula for, 67
addnode function, 443
address of (&) operator, 46, 182–184
 accessing structure members, 413
 arrays of structures, 416
 arrays, 252
 initializing variable pointer, 243
 omitting the &, 185
 referencing pointer address, 184, 245
 scanf function, 46, 47, 182

addresses
adding integer value to pointers, 253
long int type, 254–255
array names referring to addresses, 251–255
arrays, 184–185
bytes, 22
memory addresses, 184
pointers, 218, 241
why addresses vary at different times, 245
alert escape sequence, 12, 13, 17, 397
alphabetic characters
function testing for, 109, 222
alphanumeric characters
function testing for, 110, 222
wide character strings, 229
ampersand (&) character
see also address of (&) operator
& bitwise operator, 116
&& (AND) logical operator, 96–97
AND (&&) logical operator, 96–97
AND (&) bitwise operator, 116
masks, 119
anonymous unions, 450
append mode
reading strings from text files, 480
writing to text files, 471
append operations
appending strings, 208–210
opening text files for, 469
arguments for functions, 11
control strings, 25
description, 327
function parameters, 301, 302
function pointers as, 335–338
global variables replacing, 342
pass-by-value mechanism, 307–309
passing arguments and return value, 309
passing arguments to functions, 303, 307
passing parameters to main(), 349, 350
pointers to structures as, 431–432, 436–439
printf function, 25
strings as macro arguments, 534–535
structures as, 430–431, 436–439
using ordinary variables as, 310–312
using pointers as, 312–313
const keyword, 313–314
variable number of arguments, 345–349
basic rules and requirements, 348
prototyping, 345
referencing arguments, 345
sentinel value, 349
va_list type, 345, 348
arithmetic comparisons, 82–96
arithmetic expressions, 45–47
arithmetic operations
binary arithmetic operations, 57
implicit type conversion, 57
performing on char data type, 59, 62–63
arithmetic operators, 31–36
addition, 32
division, 32
modulus, 32
multiplication, 32
operator order of precedence, 103
subtraction, 32
table of, 32
working with complex numbers, 70
arithmetic statements, 31–37
arithmetic assignment statement, 25
arrays, 175–202
accessing array elements, 260–262
pointer expressions for, 260
accessing elements of, 178
adding integer value to pointers, 253
long int type, 254–255
addresses, 184–185
array names referring to addresses, 251–255
arrays as union members, 450
arrays of function pointers, 333–335
arrays of pointers, 273–283
arrays of strings, 210–212
arrays of structures, 414–416
averaging numbers with arrays, 178–180
benefits of programming with arrays, 178
changing address referenced by array name, 252
char arrays wasting memory, 206
declaring, 177
description, 177–178, 251
dimension, 177
element numbering, 177
finding size of arrays, 186–187
formatted input from arrays, 405
formatted output to arrays, 404–405
getting all values in 2D array, 257–258
index value, 177
initializing, 186
multidimensional arrays, 189–190
multidimensional arrays, 187–190
organization of arrays in memory, 185

pointers and, 251–255
multidimensional arrays, 259–262
two-dimensional arrays, 255–258
referencing, 257
retrieving elements in array, 180–181
sprintf function, 404–405
sscanf function, 405
storing strings, 205
tic-tac-toe game, 194–201
ASCII character code values, 565–569
asin function, 72
assert macro, 543–545
assert.h header file, 543
assignment (=) operator, 32
confusing with equal to (= =) operator, 82
copying strings, 208
description, 25
implicit type conversions in, 58–59
operator order of precedence, 103
shortcuts (+=, -=, *=, /=, %=), 70, 71
associativity
operator precedence, 102
asterisk (*) character
see also indirection operator (*)
see also multiplication operator
input format specifiers, 377, 576
at mode, 484
atan function, 72
atof/atoi/atol functions, 227, 350
calculator program using pointers (atof), 287, 288
atoll function, 227
automatic type conversion
see implicit type conversion
automatic variables
declaration, 340, 342
description, 296, 338
scope, 338
average
calculating average of values, 142–144, 175–177
averaging numbers with arrays, 178–180
average function
example using functions, 307
function declarations, 309, 310
pass-by-value example, 308
referencing variable number of arguments, 345–348

B

b (\b) escape sequence, 12, 18, 397
backslash (\) character
basic C syntax, 5
escape sequence for, 6, 204, 396
escape sequence to insert, 13
file path string, 471
significance in text strings, 11
backspace
escape sequence for, 12, 18, 397
bar chart
drawing vertical bar chart, 454–464
basic input/output system (BIOS), 21
beep escape sequence, 12, 13, 17, 397
best_move function, 368
big-endian system, 561–562
binary arithmetic operations, 57
binary files
see under files
binary mode
binary file input and output, 485–495
specifying, 486
text mode compared, 485, 486
binary numbers, 557–558
1's complement, 561
2's complement, 561
adding binary values, 558
negative binary numbers, 560–561
values in decimal and hexadecimal, 559
binary operators, 36
binary streams, 374
binary trees, 439–447
creating, 441–443
left node, 440
nodes, 439
ordering data in, 441
right node, 440
root node, 439
sorting data using, 444–447
traversing, 443–447
BIOS (basic input/output system), 21
bit patterns
char data type, 59
hexadecimal constants, 41
unsigned char type, 59
bits, 22, 181
bit-fields in structures, 429–430
decimal equivalents of 8-bit binary values, 558

Find it faster at http://superindex.apress.com

memory addresses, 184
sign bit, 560
bitwise operators, 116–122
logical operators and, 117
masks, 119, 121–122
op= form, 119
operator order of precedence, 103
selecting multiple bits from variables, 121–122
table of, 116
using, 119–122
blank characters
function testing for, 110, 222
wide character strings, 229
body
function body, 10, 300
defining function inside, 301
_Bool keyword, 66
bool type, 82
_Bool variable
expressions involving comparison operators, 82
if (something) expression, 84
memory-test game program, 164
operands of AND (&&) operator, 96
operands of OR (||) operator, 97
Boolean data type, 66
Boolean expressions, 82
boundary alignment, 419
effect on memory allocation, 420
boxes
drawing a box, 134–135
braces {}
basic C syntax, 5
function body, 10
functions and, 9
indenting, 10
break statement
for loop, 141–144
nested loops and goto statement, 156
switch statement, 107, 109
breakpoints, setting, 539
bubble sort, 283
buffers
buffered input operations illustrated, 485
fflush function, 170, 389
file-viewer program, 525
fputc function, 473
input and output, 170
keyboard buffer, 170
reading binary files, 487, 493, 494
reading text of arbitrary length from keyboard, 232
stderr/stdout streams compared, 375
writing to text files, 484
bugs
see debugging
Build option, 3
built-in functions, 299
bytes, 22, 182
addresses, 22
big-endian system, 562
computer affecting bytes used, 39
little-endian system, 562
memory used storing variables, 37
sizeof operator, 52
determining bytes occupied, 53

C

c conversion type
char as character and integer, 61
character output with printf function, 60
input format specifiers, 378, 576, 577
output format specifiers, 396, 575
performing arithmetic on char type, 63
reading characters using scanf(), 386
wide character conversion, 377
c extension, 5
c flag, 2
C programs
see also programming basics
basic syntax, 5
creating, 4–5
developing, 13–14
using empty functions in, 10
editing, 5–6
elements of, 7–13
ending, 350–351
indenting braces, 10
outputting information, 10–11
reserved words, 571
stages in creation of, 1
substitutions in source code, 531–532
calculator
writing program using pointers, 283–293
writing simple program, 122–126
calloc function, 268
carriage return
escape sequence for, 12, 167, 169, 397, 406

case
converting uppercase to lowercase, 93–96, 98–99
case statement
see switch statement
casting, 56
Boolean data type, 66
explicit type conversion, 57
imaginary values to complex type, 68
indirection operator (*), 242
CD-ROM drives
input and output streams, 374
ceil function, 71
change function
using ordinary variables as arguments, 311–312
using pointers as arguments, 313
char array
arrays of strings, 211
searching strings for specified character, 219
char data type, 59–60
a character or an integer, 61
arrays of strings, 210–212
char arrays wasting memory, 206
character constant, 60
common function return types, 303
comparison operators comparing, 93
conversion between uppercase and lowercase, 63
finding length of string, 207
getchar function, 272
hexadecimal value for, 62
initializing with integer value, 60
input conversion characters, 378
input format specifiers, 577
input with scanf function, 60
numeric data types, 59
output conversion characters, 396
output format specifiers, 575
output with printf function, 60
performing arithmetic on, 59, 62–63
pointer to char variable type, 271
putchar function, 404
range limits for, 50
reading characters using scanf(), 386
string and text handling methods, 205
CHAR_MIN/CHAR_MAX
range limits for integer types, 50
character constant, 60
L modifier, 96
character data types
char data type, 59–60
wchar_t data type, 63–64
character streams, 374
formatted output to screen, 401–403
getchar function, 389–390
reading character from keyboard without displaying, 389
characters
ASCII character code values, 565, 569
character classification functions, 222
character conversion functions, 224–226
control characters, 11–13
end-of-file character, 472
finding length of string, 206
input and output, 60–63
searching strings
for specified character, 219–220
wide character strings, 228
testing and converting wide characters, 229–230
whitespace characters, 396
wide character classification functions, 229
wide character strings, 227–230
check function
reading binary files, 493
cimag/cimagf/cimagl functions
complex numbers, 68
circle
calculating circumference and area, 45
clock ticks, 545
clock function, 545
memory-test game program, 167, 169
using time functions, 547
clock, processor clock, 549
clock_t type, 545, 548
CLOCKS_PER_SEC symbol, 167, 545
code
see source code
code portability
size_t integer type, 213
comma operators
operator order of precedence, 103
command line
running C program, 5
comments, 7–8
need for, 173
comparing strings, 215–218

Find it faster at http://superindex.apress.com

comparison operators, 82, 92–96
!= (not equal to) operator, 92
< (less than) operator, 82
<= (less than or equal to) operator, 92
> (greater than) operator, 82
>= (greater than or equal to) operator, 92
= = (equal to) operator, 82
else-if statement, 106–107
expressions involving, 82
if statement, 82–86
nested ifs, 89–92
if-else statement, 86–88
logical expressions, 82
operator order of precedence, 103
comparisons
arithmetic comparisons, 82–96
conditional operator, 99–101
logical operators combining, 96–99
with AND (&&) operator, 96–97
with NOT (!) operator, 97–99
with OR (||) operator, 97
compilation time
recording when program last compiled, 538
compilers
description, 2
determining bytes occupied by numeric types, 53
determining range limits for data types, 50
header files, 8
list of errors, 6
compiling
creating C program, 5
source code, 2
complement operator, 116
complex numbers, 67–70
_Complex keyword, 68, 70
_Complex_I keyword, 68
_Imaginary keyword, 68
casting imaginary values to complex type, 68
cimag function, 68
cimagf function, 68
cimagl function, 68
complex equivalent of _Complex, 68
conj function, 68
conjf function, 68
conjl function, 68
creal function, 68
crealf function, 68
creall function, 68
data types, 80
imaginary equivalent of _Imaginary, 68
polar representation, 67
types storing, 68
working with, 69–70
complex.h header file, 68, 80
exploring contents of, 70
function types included in, 351
computer_move function, 365, 369
conditional compilation
logical preprocessor directives, 536–537
conditional operator, 99–101
nested example, 100
operator order of precedence, 103
conio.h header file, 389
conj/conjf/conjl functions
complex numbers, 68
conjugate operator
complex number formula for, 67
const keyword
const parameters, 313–314
fixing variable as constant value, 49
pointers to structures as function arguments, 432
using pointers, 276
wide character strings, 228
constant expression, 55
constants
#define preprocessor directive, 48
character constant, 60
constant pointers, 251, 327
defining, 48
defining standard messages, 206
fixing variable value, 49
pointers to constants, 250
pointers to constants and constant pointers compared, 314
range limits for floating-point types, 51
range limits for integer types, 50, 51
specifying integer constants, 40–41
continuation condition
for loop, 135
continue statement, 160
control characters, 11–13
see also escape sequences
function testing for, 110, 222
wide character strings, 229
control expressions
for loop, 131, 135

control strings
see format control strings/format specifiers
conversion characters/types
description, 377
input, 377, 378, 577
output, 396, 574
conversion flags
output format specifiers, 573
conversion specifiers, 26
see also format control strings/format specifiers
conversions
explicit type, 57
implicit type, 57–59
copying strings using strcpy function, 212
copying strings using strncpy function, 213
cos/cosh functions, 72
creal/crealf/creall function
complex numbers, 68
createnode function, 442
ctime function, 549
ctype.h header file, 95, 109, 110, 352
examining strings, 221
function types included in, 351
writing calculator program using pointers, 287
curly brackets {}
see braces {}
current position
files, 468
rewind function, 482

D

d conversion type, 26
accessing values through pointers, 245
adding integer value to pointers, 254
char as character and integer, 61
character output with printf function, 60
hd format specifier, 47
input format specifiers, 378, 577
ld format specifier, 47, 56
output format specifiers, 396, 574
outputting two-character field, 179
performing arithmetic on char type, 63
reading data, 47
reading hexadecimal and octal values, 385
data
improving speed of data transfer, 419
ordering data in binary tree, 441
sorting data using binary tree, 444–447
structuring data, 409–464
data types
see also types
char, 59–60
clock_t, 545
complex, 67–70, 80
defining, 451–453
double _Complex, 68
float _Complex, 68
floating-point, 42
fpos_t, 496
integer, 38–41
long double _Complex, 68
numeric, 59–70
pointers, 242
single statement declaring multiple variables, 30
sizeof operator determining bytes occupied by, 53
time_t, 548
tm structure, 549
unsigned integer, 38–39
va_list, 345
variable declaration, 25
wchar_t data, 63–64
__DATE__ macro, 538
date and time
clock function, 545
ctime function, 549
difftime function, 546
function library, 545–555
getting the date, 549–555
getting the day for a date, 553
gmtime function, 550
localtime function, 549, 550
mktime function, 552
recording when program last compiled, 538
using time functions, 546–548
DBL_DIG, 50
DBL_MIN/DBL_MAX, 51
debugging, 538–545
description, 14
inspecting program elements, 539
integrated debuggers, 539
setting breakpoints, 539
setting watches, 539
tracing program flow, 539
using preprocessing in, 539–543
decimals
see floating-point numbers

Find it faster at http://superindex.apress.com

decision making in programs
description, 81
else-if statement, 106–107
goto statement, 115–116
if statement, 82–86
if-else statement, 86–88
switch statement, 107–115
declarations
declaring pointers, 242–243
function declarations, 309–310
variable declaration, 25
decrement operator (--), 137–138
description, 130
varying use of loop counter, 140
default keyword
switch statement, 108
#define preprocessor directive, 48
#if defined directive, 536–537
joining two results of a macro expansion, 535
macro substitutions, 532
macros that look like functions, 532–534
strings as macro arguments, 534–535
substitutions in program source code, 531–532
using in debugging, 541
delimiters
^ character specifying, 378
dereference operator
see indirection operator (*)
dereferenced pointers, 218, 246
dereferencing NULL pointer, 220
designing C programs, 14
calculating tree height program, 72
developing C programs, 13–14
see also program examples
difftime function, 546
DIG values
decimal digits for floating-point types, 50
digits (0-9)
counting digits entered from keyboard, 222
function testing for, 222
wide character strings, 229
digits (0-9, A-F hex)
function testing for, 110, 222
wide character strings, 229
dimension, arrays, 177
directives
see preprocessor directives
discounts, calculating
using if-else statement, 87
using nested conditional operators, 100
disk drives
input and output streams, 374
division
floating-point variables, 43–45
specifying decimal places, 44
specifying output field width, 44
using float variable type, 43
division by zero
calculator program example, 123
division operator, 32
complex number formula for, 67
example demonstrating, 35–36
/= shortcut, 70
negative operand(s), 36
do-while loop, 157–159
memory-test game program, 162, 166
reversing digits of integer example, 158–159
tic-tac-toe game, 197
typical usage, 173
double _Complex type, 68, 80
double _Imaginary type, 68, 80
double precision floating-point numbers, 42
double quotes
escape sequence for, 13, 204
string constants, 203
double variable type, 42
atof function, 227
common function return types, 304
format specifier for, 47
implicit type conversion rules, 58
input type conversion modifiers, 377
range limits for, 51
doubly linked list
pointers to structures as structure members, 426–429
duplicatefile function, 508
updating exiting file, 515
dynamic memory allocation, 263–271
allocating small amounts of memory, 271
calculating prime numbers, 264–268
calloc function, 268
checking memory available, 263
free function, 269–270
guidelines for working with memory, 271
malloc function, 263–268
memory leak, 268

reallocating memory, 270–271
releasing memory, 268–270, 271
sizeof operator, 264
structures, 418–420

E

E conversion type, 396, 575
exponential for floating-point values, 41, 563
e conversion type, 52, 575
input format specifiers, 378
le format specifier, 47
output format specifiers, 396
reading data, 47
variations on floating-point input, 384
eatspaces function, 393
editing source code
editing simple C program, 5–6
emacs editor for Windows, 2
programming basics, 1–2
UNIX/Linux, 2
using word processors, 2
elements
inspecting program elements, 539
elements, arrays
addresses, 184
description, 177
initializing, 186
numbering, 177
pointer expressions for accessing, 260
referencing, 257
retrieving elements in array, 180–181
#elif preprocessor directive, 538
ellipsis
prototyping functions, 345
#else preprocessor directive, 537–538
else keyword
else-if statement, 106–107
if-else statement, 86–88
emacs editor, 2
#endif preprocessor directive, 536
using in debugging, 540, 541
ending programs, 350–351
end-of-file character, 472
enum keyword, 64
enumeration constants, 64
enumeration type, 304
enumerations, 64–66
controlling switch statement with, 108
integer types, 64
unnamed enumeration types, 66
enumerators, 64
choosing enumerator values, 65
EOF character, 472
feof function, 484
writing and reading files, 476
EOR bitwise operator, 116, 117
equal to (= =) operator, 82
comparing strings, 216
complex numbers, 67
confusing with assignment (=) operator, 82
error messages
stderr stream, 375
errors, 6, 483–484
see also debugging
common mistakes, 18
executing programs, 3
feof function, 484
ferror function, 484
missing semicolon (;), 6
perror function, 483
escape sequences, 11–13
see also control characters
\\, 13
\', 13
\", 13
\?, 13
\a, 12, 13
\b, 12
\f, 12
\n, 11, 12
\r, 12
\t, 12
\v, 13
alert, 12, 13
backslash, inserting in text string, 13
backspace, 12
beep, 12, 13
carriage return, 12
description, 6
double quote, 13
form-feed, 12
horizontal tab, 12
newline, 11, 12
null character, 204
printf()/wprintf(), 396–397
question mark, 13
quotation marks, 13
significance of \ character in text strings, 11
single quote, 13

Find it faster at http://superindex.apress.com

strings, 204
vertical tab, 13
exe extension, 3
executing C program, 5
executing programs
creating C program, 5
errors created when, 3
programming basics, 3
start point for program execution, 9
exit function, 320, 351
dealing with errors, 483
exp function, 71
explicit type conversion, 57
exponents
floating-point numbers, 563
floating-point values, 41
output conversion characters, 396
output format specifiers, 575
variations on floating-point input, 384
expressions
return values, functions, 301
structures in expressions, 417
extern keyword, 531
external devices
using as memory, 467
external files
including header files in programs, 530

F

f (\f) escape sequence, 12
common escape sequences, 397
writing to printers, 406
F conversion type, 575
f conversion type, 43, 52, 575
calculating circumference/area of circle, 46
float constant, 42
input format specifiers, 378
lf format specifier, 47
output format specifiers, 396
pass-by-value example, 308
reading data, 47
variations on floating-point input, 384
fabs function, 71
factorial function, 343, 344
factorials
calculating factorial of integer, 343–345
fclose function, 472, 476
feof function, 484, 507, 512
ferror function, 484
fflush function, 170, 389, 472
switching from read to write, 485
updating binary files, 506
fgetc function, 474
fgetpos function, 496
fgets function, 223, 224, 477
character conversion example, 226
gets() compared, 281
random access to files, 500
reading strings, 389
reading strings from text files, 480
reading text of arbitrary length from keyboard, 231, 232, 233
sorting strings using pointers, 281, 282
writing calculator program using pointers, 285
fgetws function, 229, 230
field width
input format specifiers, 377
output format specifiers, 396
specifying in floating-point division, 44
file modes, 469
file path string
escaping backslash characters, 471
file pointers, 468, 469, 470
FILE structure, 470
file-viewer
writing file-viewer program, 522–527
FILENAME_MAX, 470
files
accessing, 468–473
accessing data in files, 495–502
random access to file, 497–502
binary files, 468
changing file contents, 508
how data written in, 468
input and output, 485–495
listing contents, 513–514
mode specifiers, 486, 521
reading, 487–495, 517
reading record from keyboard, 509–510
reading record from file, 511–512
updating, 503–517, 521
updating existing file contents, 514–517
writing, 486–487, 512–513, 517
writing record to file, 510–511
closing, 472
creating unique file name, 502–503
current position, 468
dealing with errors, 483–484

deleting, 472
EOF character, 472
functions
duplicatefile, 508, 515
fclose, 472
feof, 484
ferror, 484
fflush, 472
fgetc, 474
fgetpos, 496
fgets, 477
findrecord, 508, 515, 517
fopen, 469–471
fprintf, 481
fputc, 473–474
fputs, 476–477
fscanf, 481
fseek, 496–497
fsetpos, 497
ftell, 496
getname, 508, 510
getrecord, 508, 509–510
listfile, 508, 513–514
perror, 483
readrecord, 508, 511–512
remove, 472
rename, 471
rewind, 482
updatefile, 514–517
writefile, 508, 512–513
writerecord, 508, 510–511
file positioning operations, 495–502
file streams, 468
FILENAME_MAX, 470
FOPEN_MAX, 470
formatted input from, 481–483
formatted output to, 481, 481–483
introduction, 467
mode specifiers, 484–485, 521
moving around in, 495–502
open files limit, 470, 472
opening, 469–471
call to fopen() fails, 470
limit to length of filename, 470
read/write/append operations, 469
position in, 468
renaming, 471
structure illustrated, 468
temporary work files, 502–503
text files, 468
reading from, 474–476
reading strings from, 477–480
writing strings to, 476–480
writing to, 473–476
using as memory, 467
findrecord function, 508
updating exiting file, 515, 517
flags
output format specifiers, 573
float _Complex type, 68, 80
float _Imaginary type, 68, 80
float constant
suffix specifying, 42
float conversion characters, 384
float variable type, 42
common function return types, 303
division using, 43
format specifier for, 47
implicit type conversion rules, 58
input conversion characters, 378
range limits for, 51
range of values for, 42
float.h header file, 50, 51
floating-point numbers/variables, 42, 562–563
%e format specifier, 52
%f format specifier, 52
accessing decimal digits for, 50
decimal notation, 557
8-bit binary values, 558
division using, 43–45
specifying decimal places, 44
specifying output field width, 44
exponentials, 41, 563
left-aligned output, 44
output conversion characters, 396
output flags, 395
output format specifiers, 575
outputting values, 400–401
precision, 563
range limits for, 51
values for, 41
values in binary and hexadecimal, 559
variations on input, 383–384
floor function, 71
FLT_DIG value, 50
FLT_MIN/FLT_MAX, 51
flushing
see fflush function

Find it faster at http://superindex.apress.com

fopen function, 469–471
 call to fopen() fails, 470
 file modes, 469
 reading binary files, 491
 return value, 469
 writing and reading files, 476
FOPEN_MAX, 470
for loop, 131–136, 138–146, 148–149
 averaging numbers with arrays, 179
 break statement in, 141–144
 calculating average of values example, 142–144
 character conversion, 224, 226
 continuation condition, 135
 continue statement, 160
 control expressions, 131, 135
 calculation in third control expression, 139–140
 floating-point loop control variables, 149
 example drawing box, 134–135
 increment operator (++), 135
 indefinite looping, 141
 limiting user input example, 144–146
 loop counter, 135
 varying use of, 140–141, 148–149
 memory-test game program, 167
 postfix increment operator, 139, 140
 prefix increment operator, 140
 retrieving elements in array, 181
 summing numbers, 138–140
 using nested for loop, 153–154
 using while loop nested inside, 154–156
 syntax, 135
 tic-tac-toe game, 199
 typical usage, 173
 without parameters, 141
format control strings/format specifiers
 % character, 26
 always displaying +/- sign, 70
 characters in input format string, 382–383
 display floating-point values, 43
 format control string, 25
 general form of, 377
 input conversion characters, 378
 input format control strings, 376–381
 input format specifiers, 377, 576–578
 integer conversion specifiers, 377
 left-aligned output, 44
 output conversion characters, 396
 output format specifiers, 395–396, 573–575
 placing characters in square brackets, 378
 printf/wprintf functions, 394, 395
 reading characters using scanf(), 386–387
 reading data, 47
 reading hexadecimal and octal values, 384–386
 scanf/wscanf functions, 376–381
 specifying decimal places in floating-point division, 44
 specifying output field width, 44
 string to include only specified characters, 386
 using wrong specifier, 43
 variations on floating-point input, 383–384
formatted file input/output, 375, 480–483
form-feed
 escape sequence for, 12, 397, 406
forward slash (/)
 basic C syntax, 5
 comments, 7
fpos_t type, 496
fprintf function
 formatted output to files, 481
 output format specifiers, 573
 writing to printers, 405–406
fputc function, 473–474
fputs function
 reading strings from text files, 479, 480
 writing strings to text files, 476–477
fread function
 reading binary files, 487
 updating binary files, 507
free function, 280, 269–270
 using linked lists, 426
freenodes function, 447
fscanf function, 481
 formatted file input and output, 483
 input format specifiers, 576
fseek function, 496–497
fsetpos function, 497, 501
ftell function, 496
function body, 10
 defining function inside, 301
 defining functions, 300
 description, 327
 functions without bodies, 300
function declarations
 see function prototypes

function headers
defining functions, 300
description, 327
examples of, 302
function parameters, 302
function pointers, 329–338
arrays of, 333–335
as arguments, 335–338
calling functions through, 330–333
declaring, 329–330
declaring arrays of, 333
function prototypes
description, 309, 327
passing data to function using pointers, 318
reading binary files, 489
functions, 299–307
arguments for, 11
const keyword, 313–314
passing arguments to functions, 303, 307
pointers to structures as, 431–432, 436–439
structures as, 430–431, 436–439
using pointers, 312–313, 315–319
using variables as, 310–312
variable number of, 345–349
built-in functions, 299
calling, 302
character classification functions, 222
character conversion functions, 224–226
converting strings to numerical values, 227, 350
declarations, 309–310
defining, 300
description, 9, 326
empty functions, 10
example using, 305–307
execution of program made up of several, 296
external variables and, 530–531
functions not returning values, 300
functions operating on wide character strings, 228
functions that call themselves, 343–345
general form of, 300, 301
hyperbolic functions, 72
incrementing pointers in, 326
inline functions, 352
interfaces for, 15
inverse trigonometric functions, 72
libraries of, 15, 351–352
macros that look like, 532–534
maintaining variable within declared scope, 338–340
mathematical functions, 71–72
modular programming and, 14–16
naming conventions, 301
not protecting original values passed to, 308, 312–313
parameters, 300, 302
pass-by-value mechanism, 307–309
performance, 352–353
pointers to functions
see function pointers
protecting original values passed to, 308, 310–312
passing pointers, 313–314
prototyping using ellipsis, 345
recursive function calls, 343–345
restrict keyword, 353
return statement, 9, 304–305
return value type, 9
specifying, 303–304
return values, 300, 301
structures as, 432–435
returning pointer values from functions, 322–326
using local storage, 323–324
sharing variables between, 340–342
string functions, 212–221
time functions, 546–548
trigonometric functions, 72
variable scope, 299
variables in functions, 338–342
wide character classification functions, 229
functions, list of
abort, 350
acos, 72
addnode, 443
asin, 72
atan, 72
atof/atoi/atol/atoll, 227
average, 307, 345, 348
best_move, 368
calloc, 268
ceil function, 71
change function, 311
check function, 493
cimag/cimagf/cimagl, 68
complex numbers, 68
conj/conjf/conjl, 68
clock, 167, 545

Find it faster at http://superindex.apress.com

computer_move, 365, 369
cos/cosh, 72
creal/crealf/creall, 68
createnode, 442
ctime, 549
difftime, 546
display, 357, 358
duplicatefile, 508, 515
eatspaces, 393
exit, 320, 351
exp, 71
fabs, 71
factorial, 343
fclose, 472
feof, 484
ferror, 484
fflush, 170, 389, 472
fgetc, 474
fgetpos, 496
fgets, 223, 224, 477
fgetws, 229
findrecord, 508, 515, 517
floor, 71
fopen, 469–471
fprintf, 405–406, 481
fputc, 473–474
fputs, 476–477
fread, 487
free, 269–270
freenodes, 447
fscanf, 481
fseek, 496–497
fsetpos, 497
ftell, 496
fwrite, 486–487
get_parent_dob, 500, 501
get_person, 432–436
get_score, 367
getc, 474
getch, 389
getchar, 272, 389–394
getinteger, 393, 394
getname, 394, 508, 510
getrecord, 508, 509–510
gets, 223, 388–389
gmtime, 550
IncomePlus, 323
isalnum/isalpha/isblank/iscntrl, 222
isdigit/isgraph/islower, 222
isnewline, 393
isprint/ispunct/isspace/isupper, 222
iswalnum/iswblank/iswcntrl/iswdigit, 229
iswgraph/iswlower/iswprint/iswpunct, 229
iswspace/iswupper/iswxdigit, 229
isxdigit, 222
listfile, 508, 513–514
listnodes, 444
localtime, 549, 550
log/log10, 71
main, 9–10, 349–350
make_move, 365
malloc, 263–268
mktime, 552, 554
perror, 483
pow, 293
printf, 10
product, 543
put_primes, 487, 491, 494
putc, 474
putchar, 404
puts, 403–404
rand, 146–148
readrecord, 508, 511–512
realloc, 270–271
related, 437
remove, 472
rename, 471
rewind, 482
scanf, 46, 376–388
set_ancestry, 436, 437
siblings, 431, 432
sin/sinh, 72
sprintf, 404–405
sqrt, 548
srand, 147
sscanf, 405
str_in, 319–320
str_out, 322
str_sort, 320–321
strcat, 214–215
strchr, 219–220
strcmp, 215–218
strcpy, 212
strlen, 208, 213
strncat, 214
strncmp, 216
strncpy, 213
strrchr, 220

strstr, 220–221
swap, 321–322
tan/tanh, 72
test_prime, 487, 490, 492
time, 147, 546
tmpnam, 503
tolower/toupper, 224
towlower/towupper, 229
ungetc, 390
updatefile, 514–517
va_arg/va_end/va_start, 345
valid_moves, 360
wcscat/wcschr/wcscmp/wcscpy, 228
wcslen/wcsncmp/wcsncpy/wcsstr, 228
wprintf, 394
writefile, 512–513
writerecord, 510–511
wscanf, 376
fwrite function
writing binary files, 486–487

G

G conversion type, 396, 575
g conversion type, 575
input format specifiers, 378
output format specifiers, 396
variations on floating-point input, 384
get_parent_dob function, 500, 501
get_person function
pointers to structures, 436
random access to files, 499
returning pointers to structures, 432–435
get_score function, 367
getc function, 474
confusing with gets(), 474
writing and reading files, 476
getch function, 389
getchar function, 272, 275, 389–394
reading strings up to given limit, 278
unformatted input, 375
getinteger function, 393, 394
getname function, 394, 510
description, 393, 508
random access to files, 499
getrecord function, 509–510
description, 508
gets function, 388–389
confusing with getc(), 474
counting digits and letters entered from keyboard, 223
description, 223
disadvantage of, 223
drawing vertical bar chart example, 463
fgets() compared, 281
sorting strings using pointers, 281
writing calculator program using pointers, 284
gigabyte, 23
global scope, 489
global variables, 340–342
declaration position making global, 340, 342
declarations of, 531
extern keyword, 531
scope, 340
using sparingly, 342
gmtime function, 550
goto statement, 115–116
nested loops and, 156–157
graphs
drawing vertical bar chart, 454–464
greater than (>) operator, 82
comparing strings, 215
greater than or equal to (>=) operator, 92

H

h (.h) extension, 8
h modifier
hd format specifier, 47
input modifiers, 377, 576
output modifiers, 396, 574, 575
hash (#) character, 8
operator, 535
flag, output specifier, 395, 574
preceding parameter in macro expansion, 534
preprocessor directives, 529
hat size program
using multidimensional arrays, 190–193
and pointers, 261–262
header files, 351–352
assert.h, 543
case sensitivity, 8
complex.h, 68, 80
conio.h, 389
ctype.h, 95
description, 8
extension indicating, 8
float.h, 50
including header files in programs, 530
limits.h, 50

Find it faster at http://superindex.apress.com

math.h, 71
NULL defined in, 243
stdarg.h header file, 345
stdbool.h, 66
stddef.h, 213
stdio.h, 8
stdlib.h, 146, 147
string.h, 212
time.h, 147
wchar.h, 228
wctype.h, 96
headers
function headers, 300, 302
heap, 263
hexadecimal constants
bit patterns, 41
hexadecimal form
integers, 40
writing file-viewer program, 522–527
hexadecimal numbers, 558–560
input conversion characters, 378, 577
output conversion characters, 396, 575
output flags, 395
reading hexadecimal and octal values, 384–386
values in decimal and binary, 559
horizontal tab
escape sequence for, 12, 17, 397
hyperbolic functions, 72

I

i conversion type
input conversion characters, 378
output conversion characters, 574
IDE (integrated development environment), 1
Build option, 3
#if defined preprocessor directive
conditional compilation, 536–537
#if preprocessor directive
conditional compilation, 536–537
directives testing for specific values, 537
if statement, 82–86
else-if statement, 106–107
if-else statement, 86–88
if (something) expression, 84
nested if statements, 89–92
tic-tac-toe game, 199
using blocks of code within, 88–89
#ifdef preprocessor directive, 537
using in debugging, 540, 541
using preprocessing in debugging, 543
#ifndef preprocessor directive, 537
_Imaginary keyword, 68
complex.h header file, 68, 80
imaginary values
see complex numbers
implementing C programs, 14
implicit type conversion, 57–59
Boolean data type, 66
in assignment statements, 58–59
rules for, 57–58
include files
see header files
#include preprocessor directive, 8, 318, 530
IncomePlus function
increment operator (++), 136–137
description, 130
for loop, 132, 135
postfix form, 137
for loop, 139, 140
while loop, 151
prefix form, 137
for loop, 140
while loop, 152
using pointers, 248
indefinite looping, 141
indentation of program code, 173
index value, arrays, 177
arrays and addresses, 184
multidimensional arrays, 188
indexes
accessing data in files, 495
indirection operator (*)
casting malloc() return value, 263
casting values, 242
declaring function pointer, 330
dereferenced pointers, 218, 220, 246
description, 243
interpreting * as multiplication or indirection, 245
levels of indirection, 260
multidimensional arrays, 260
pointer expressions for accessing array elements, 260
pointers to structures as function arguments, 432
two-dimensional arrays, 256, 257
using, 245

initializing variables, 30–31
inline functions, 352
input
 see also read operations
 binary file input and output, 485–495
 characters, 60–63
 scanf function, 46
input format specifiers, 377, 576–578
 see also format control strings/format specifiers
input streams
 characters in input format string, 382–383
 description, 373
 format control strings, 376–381
 formatted input, 375
 from arrays, 405
 from files, 481–483
 from keyboard, 376–388
 getchar function, 389–394
 gets function, 388–389
 input from keyboards, 375–394
 reading any sequence of characters, 378, 380
 reading any sequence of consecutive digits, 387
 reading characters using scanf(), 386–387
 reading first non-space character, 387
 reading hexadecimal and octal values, 384–386
 scanf function, 373, 376–388
 sscanf function, 405
 standard and nonstandard streams, 374
 standard streams, 374–375
 string input from keyboard, 388–389
 strings including spaces, 386
 unformatted input, 375
 from keyboard, 389–394
 variations on floating-point input, 383–384
int keyword, 25
int variable type
 atoi function, 227
 common function return types, 303
 format specifier for, 47
 input conversion characters, 378
 output conversion characters, 396
 range limits for, 50
 range of values, 38
 suffix requirement, 39
INT_MIN/INT_MAX, 50
integer conversion specifiers
 input format specifiers, 377
integers, 24–27
 abbreviations, 38
 avoiding leading zeros, 41
 calculating factorial of, 343–345
 char data type, 59
 computer affecting bytes used, 39
 description, 24
 enumerations, 64
 example using, 24, 26, 29
 examples of integer numbers, 24
 examples of noninteger numbers, 24
 generating pseudo-random integers, 146–148
 getinteger function, 393, 394
 hexadecimal form, 40
 implicit type conversion rules, 58
 input format specifiers, 577
 integer output, 397–399
 integer variable types, 38–41
 memory usage, 38
 output format specifiers, 574
 range limits for, 50
 outputting values, 51
 reversing digits of integer example, 158–159
 size_t integer type, 52
 specifying integer constants, 40–41
 L/LL/U/UL suffixes, 40
 summing numbers using for loop, 138–140
 summing numbers using nested for loop, 153–154
 summing numbers using while loop, 151–152
 summing numbers using while loop nested inside for loop, 154–156
 unsigned integer types, 38–39
 using integer types, 39–40
integrated development environment (IDE), 1, 3
integration testing, 14
interfaces, functions, 15
inverse trigonometric functions, 72
isalnum function, 110, 222, 233
isalpha function, 109, 222
isblank/iscntrl/isdigit/isgraph/islower functions, 110, 222
isnewline function, 393
ISO/IEC standard libraries, 351

Find it faster at http://superindex.apress.com

isprint function, 110, 222, 403, 524
ispunct function, 110, 222
isspace function, 110, 222, 390
isupper function, 110, 222
iswalnum/iswblank/iswcntrl/iswdigit functions, 229
iswgraph/iswlower/iswprint/iswpunct functions, 229
iswspace/iswupper/iswxdigit functions, 229
isxdigit function, 110, 222

J

join() macro, 535
justified values, output flags, 395

K

keyboard buffer, 170
keyboards
 formatted input from, 376–388
 input from, 375–394
 reading character from without displaying, 389
 reading record from, 509–510
 string input from, 388–389
 unformatted input from, 389–394
keywords, 9
 _Bool, 66
 _Complex, 68
 _Complex_I, 68
 _Imaginary, 68
 const, 49
 continue, 160
 description, 25
 extern, 531
 floating-point variable types, 42
 inline, 352
 int, 25
 integer variable types, 38
 list of reserved words, 571
 long, 40
 naming functions, 301
 restrict, 353
 short, 40
 sizeof, 52–53
 static, 339
 struct, 409
 typedef, 451–453
 unsigned, 38
 void, 9
kilobyte, 23

L

L modifier
 character constants, 96
 input modifiers, 377, 576
 long double constant, 42
 long double types, 52
 long int constant, 39, 40
 nested if example, 91
 output modifiers, 396, 574, 575
 unsigned long int constant, 40
 wide character strings, 227, 230
l modifier
 input modifiers, 377, 576
 integer output, 398
 ld format specifier, 47, 56
 le/lf format specifiers, 47
 output modifiers, 396, 574, 575
LDBL_DIG, 50
LDBL_MIN/LDBL_MAX, 51
left node, binary trees, 440
left-aligned output, 44, 63
left-justified values, output flags, 395, 574
less than (<) operator, 82
 comparing strings, 215
less than or equal to (<=) operator, 92
letters
 counting letters entered from keyboard, 222
levels of indirection, 260
libraries
 date and time function library, 545–555
 function libraries, 15, 351–352
 header files contained in compilers, 8
 ISO/IEC standard libraries, 351
 linking program files, 3
 string library functions, 212–221
lifetime, variables, 297
limits.h header file, 50, 51
linked list
 description, 426
 doubly linked list, 426–429
 pointers to structures as structure members, 422–426
linker, 3
linking, 3, 5
listfile function, 513–514
 description, 508
 updating binary files, 506
listnodes function, binary tree, 444, 447
lists
 copying va_list type, 348

literals
L modifier, 230
little-endian system, 561–562
ll modifier
input format specifiers, 576
input type conversion modifiers, 377
output format specifiers, 574, 575
output type conversion modifiers, 396
LL suffix
long long int constant, 40
LLONG_MIN/LLONG_MAX, 50
local storage
returning pointer values from functions using, 323–324
local variable
caution returning address of local variable, 324
localtime function, 549, 550
getting the date, 550, 552
getting the day for a date, 553, 555
log/log10 functions, 71
logical expressions, 82
logical operators, 96–99
AND (&&) operator, 96–97
bitwise operators and, 117
NOT (!) operator, 97–99
operator order of precedence, 103
OR (||) operator, 97
selection for interview example, 104–106
logical preprocessor directives, 536–538
conditional compilation, 536–537
directives testing for specific values, 537
multiple-choice selections, 537–538
standard preprocessing macros, 538
long _Complex type, 80
long double _Complex type, 68
long double _Imaginary type, 68, 80
long double constant, 42
long double variable type, 42
common function return types, 304
implicit type conversion rules, 58
input type conversion modifiers, 377
output type conversion modifiers, 396
range limits for, 51
range of values for, 42
long int constant, 39, 40
long int variable type
adding integer value to pointers, 254–255
atol function, 227
common function return types, 303
format specifier for, 47
input type conversion modifiers, 377
output type conversion modifiers, 396
range limits for, 50
range of values, 38
using, 40
long keyword, 40
long long int constant, 40
long long int variable type
atoll function, 227
common function return types, 303
input type conversion modifiers, 377
output type conversion modifiers, 396
range limits for, 50
range of values, 38
LONG_MIN/LONG_MAX, 50, 91
loop counter
for loop, 135, 140–141, 148–149
loops, 129–173
continue statement, 160
decrement operator (--), 130, 137–138
determining frequency of words in text, 234
do-while loop, 157–159
ending loop with \n or \0, 272
for loop, 131–136, 138–146, 148–149
how loops work, 129
increment operator (++), 130, 136–137
indefinite looping, 141
memory leaks, 271
nested loops, 153–157
goto statement and, 156–157
recursive functions, 343
while loop, 149–152
lowercase
function testing for, 110, 222
wide character strings, 229
lowercase letters
converting upper/lower cases, 93–96, 98–99
wide character strings, 229

M

macros
assert() macro, 543–545
join() macro, 535
joining two results of a macro expansion, 535
macro substitutions, 532
macros that look like functions, 532–534
multiple macro substitutions, 536
standard preprocessing macros, 538
strings as macro arguments, 534–535

Find it faster at http://superindex.apress.com

main function, 9–10
 calling through function pointer, 332
 declaring function pointer, 332
 description, 326, 508
 ending programs, 350
 example using, 17
 example using functions, 306
 modular programming, 15
 passing parameters to, 522, 523, 349–350
 program structure, 295
 writing file-viewer program, 522
make_move function, 365
malloc function, 263–268
 casting return value, 263
 description, 419
 dynamic memory allocation with structures, 419, 420
 memory leaks, 325
 pointer to void variable type, 303
 reading strings up to given limit, 279, 280
 releasing memory, 268
 returning pointers to structures, 435
 sorting strings using pointers, 282
 using linked lists, 425
 writing binary files, 487
masks
 AND bitwise operator, 119
 bitwise operators, 119, 121
 OR bitwise operator, 120
 selecting multiple bits from a variable, 121–122
math.h header file, 71
 function types included in, 351
 writing calculator program using pointers, 293
mathematical functions, 71–72
 asin/acos/atan, 72
 ceil, 71
 cos/cosh, 72
 exp, 71
 fabs, 71
 floor, 71
 hyperbolic functions, 72
 inverse trigonometric functions, 72
 log/log10, 71
 pow, 71
 sin/sinh, 72
 sqrt, 71
 tan/tanh, 72
 trigonometric functions, 72
MAX values
 range limits for floating-point types, 51
 range limits for integer types, 50
MAXLEN, 506
MAXWORDS, 233, 234
megabyte, 23
members, structures
 accessing structure members, 411–414
 assigning values to members, 421
 description, 410
 member selection operator, 411
 pointer to member operator, 417
 pointers to structures as, 422–429
 structures as members of structures, 420–421
 structures declared within structures, 421–422
memory, 21–23, 181–182
 address of (&) operator, 182–184
 basic input/output system (BIOS), 21
 bits, 181
 byte based measurements of, 23
 bytes, 182
 calloc function, 268
 char arrays wasting, 206
 dynamic memory allocation, 263–271
 releasing memory, 268–270
 sizeof operator, 264
 structures, 418–420
 effect of boundary alignment on allocation of, 420
 guidelines for working with, 271
 heap, 263
 improving data transfer with processor, 419
 integer variable types, 38
 keyboard buffer, 170
 malloc function, 263–268
 memory addresses, 184
 memory used storing variables, 37
 multidimensional arrays, 188
 organization of an array in, 185
 pointers, 218, 241
 random access memory (RAM), 21, 22
 read-only memory (ROM), 21
 reallocating, 270–271
 sharing memory between variables, 447–451
 unions, 448–451
 stack, 263
 structures, 410
 using linked lists, 426

using bit-fields in structures, 429–430
using files as, 467
variables and, 23
memory leaks, 268, 325
loops, 271
memory-test game program, 160–172
messages
defining standard messages, 206
varying output message, 101
MIN values
range limits for floating-point types, 51
range limits for integer types, 50
minimum/maximum of variables, 99
minus operator
see also subtraction operator
unary minus operators, 36
mktime function, 552, 554
modes
file mode specification, 469
file modes, 469
text modes, 484–485
updating binary files, 503
modifier value, %f format specifier, 44
modular programming
function libraries, 15
functions and, 14–16
reasons for using, 15
testing, 14
modulus operator
%= shortcut, 70
arithmetic operators, 32
example demonstrating, 35–36
negative operand(s), 36
multidimensional arrays, 187–188
see also two-dimensional arrays
accessing array elements, 260–262
example using, 190–193
indirection operator (*), 260
initializing, 189–190
pointers and, 259–262
referencing arrays/subarrays/elements, 257
tic-tac-toe game, 194–201
multiplication operator (*), 32
*= shortcut, 70
complex number formula for, 67
example demonstrating, 33–34
interpreting * as multiplication or indirection, 245

N

n (\n) escape sequence, 11, 12
common escape sequences, 397
example using, 17
printf function, 404
puts function, 404
n conversion type
input format specifiers, 378, 577
output format specifiers, 575
naming conventions
functions, 301
structures, 410
naming pointers, 251
naming variables, 27–29
automatic/static/global variables, 342
NDEBUG symbol, 544
negative binary numbers, 560–561
nested blocks, 301
nested if statements, 89–92
nested loops, 153–157
goto statement and, 156–157
newline character, 11
determining frequency of words in text, 233
escape sequence for, 11, 12, 17, 397, 404
including in strings, 204
isnewline function, 393
writing to printers, 406
nodes
binary trees, 439
creating, 441–443
ordering data in, 441
traversing, 443–447
left node, 440
right node, 440
root node, 439
NOT (!) logical operator, 97–99
operator precedence, 103
when not to use, 173
NOT bitwise operator, 116, 120
not equal to (!=) operator, 92
noughts and crosses game, 194–201
NULL, 147
header files defined in, 243
null character
affecting length of strings, 204
ASCII character code values, 569
finding length of string, 207
marking end of strings, 204
outputting a null-terminating string, 206

Find it faster at http://superindex.apress.com

NULL pointer, 243
dereferencing, 220
header files containing definition, 250
testing for NULL pointer, 250
numbers
binary, 557–558
negative binary, 560–561
floating-point, 562–563
hexadecimal, 558–560
numeric data types, 59–70
calculating tree height program, 72–79
implicit conversion in assignment statements, 58–59
sizeof operator determining bytes occupied by, 53
numerical values
converting strings to, 227, 350

O

o (.o) file extension, 2
o conversion type
reading octal values, 384–386
input format specifiers, 378, 577
output format specifiers, 396, 574
obj (.obj) file extension, 2
object code, 2
object files, 2
octal numbers, 41
input conversion characters, 378
input format specifiers, 577
output conversion characters, 396
output flags, 395
reading hexadecimal and octal values, 384–386
op= shortcuts (&=, |=, ^=, >>=, <<=)
bitwise operators, 119
op= shortcuts (+=, -=, *=, /=, %=)
assignment operations, 70–71
operands, 36
binary arithmetic operations, 57
implicit type conversion rules, 58
operations
assignment, 32
file positioning, 495–502
operators
arithmetic, 31–36
assignment, 25
binary, 36
bitwise, 116–122
comparison, 82, 92–96
conditional, 99–101
decrement operator (--), 130, 137–138
dereference operator, 218
increment operator (++), 130, 136–137
logical, 96–99
AND (&&) operator, 96–97
NOT (!) operator, 97–99
OR (||) operator, 97
member selection operator (structures), 411
operator precedence, 102–106
associativity determining, 102
pointer to member operator, 417
sizeof operator, 52–53, 186–187
ternary operator, 99
unary minus operator, 36–37
unary operator, 36
OR (||) logical operator, 97
OR bitwise operator, 116, 117
masks, 120
order of operator precedence, 102–106
need to understand, 173
Othello
see Reversi program
output
binary file input and output, 485–495
characters, 60–63
outputting two-character field, 179
retrieving elements in array, 181
varying output message, 101
output field width
specifying in floating-point division, 44
output format specifiers, 395–396, 573–575
see also format control strings/format specifiers
always displaying +/- sign, 70
output streams
description, 373
formatted output to arrays, 404–405
formatted output to files, 481–483
formatted output to screen
character output, 401–403
escape sequences printf()/wprintf(), 396–397
integer output, 397–399
outputting floating-point values, 400–401
using printf(), 394–396
formatted output to screens, 394–403
fprintf function, 405–406
printf function, 11, 373, 394–403
putchar function, 404

puts function, 403–404
sprintf function, 404–405
standard and nonstandard streams, 374
standard streams, 374–375
unformatted output to screen, 404
wprintf function, 401–403
writing to printers, 405–406

P

p conversion type
address of (&) operator, 184
output format specifiers, 575
outputting variable address, 245
p prefix
naming pointers, 251
padding with zeros
output flags, 395
PAGE_LENGTH, 525, 526
parameters
const parameters, 313–314
description, 300
function parameters, 302
reading binary files, 489
separating parameters, 302
multiple parameterized substitutions, 532
passing to main(), 349–350, 522, 523
parentheses
arithmetic expressions, 45–47
declaring pointer to function, 330
function parameters, 300
operator order of precedence, 103
pass-by-value mechanism, 307–309, 327
percent (%) character
%% escape sequence, 396, 399
%[] format specifier, 376, 386, 389
%p format specifier, 245
conversion specifiers, 26
escape sequence for, 396
input format specifiers, 377, 576
output format specifiers, 395, 573, 575
performance
declaring inline functions, 352
enhancing performance, 352–353
using restrict keyword, 353
perror function, 483
placeholders, 300, 302
plural words
outputting appropriate singular/plural form, 101
pointer to char variable type, 271
pointer to member operator, 417
pointer to void variable type, 303
pointer to xyz types
common function return types, 303
pointers, 218, 241–294
accessing values through, 243–246
adding integer value to, 253
long int type, 254–255
address of (&) operator, 245
addressing multiple variables with, 246
array names referring to addresses, 251–255
arrays and, 251–255
accessing array elements, 260–262
multidimensional arrays, 259–262
two-dimensional arrays, 255–258
calloc function, 268
caution returning address of local variable, 324
caution uninitialized pointers, 280
constant pointers, 251
pointers to constants compared, 314
declaring, 242–243, 244–245
dereference operator, 218
dereferenced pointers, 246
description, 251
determining pointer type, 319
dynamic memory allocation, 263–271
sizeof operator, 264
enhancing performance, 353
free function, 269
function pointers, 329–338
handling strings using, 271–283
reading text with more control, 272
using arrays of pointers, 273–283
incrementing in functions, 326
indirection operator (*), 243
initializing, 243, 248
introduction to, 241–242
levels of indirection, 260
malloc function, 263–268
naming, 251
not protecting original values passed to, 312–313
not protecting original values passed to functions, 308
NULL, 243
obtaining number of bytes occupied, 245
output format specifiers, 575
passing data to function using, 315–319
pointers to constants, 250
constant pointers compared, 314

Find it faster at http://superindex.apress.com

pointers to structures, 417
as function arguments, 431–432, 436–439
as structure members, 422–429
doubly linked list, 426–429
dynamic memory allocation, 418–420
linked list, 422–426
returning, 432–435
simplifying code using typedef, 453
structures as function return value, 432–435
pointers to unions, 450
protecting original values passed to, 313–314
releasing memory, 268–270
returning pointer values from functions, 322–326
using local storage, 323–324
searching strings for specified character, 219
searching strings for substrings, 220
searching strings for wide character, 228
sizeof operator, 245
sorting strings using, 280–283
specifying return value type, 303
swap function, 322
testing for NULL pointer, 250
using, 246–250
using with scanf(), 249
variable types, 242
variables, 241
variable/pointer relationship, 249
void type, 242
writing calculator program using, 283–293
polar representation, complex numbers, 67
postfix form, increment operator, 137
for loop, 139, 140
while loop, 151
pow function, 71
writing calculator program using pointers, 293
precedence
operator precedence, 102–106
precision format specifier, 396, 574
precision value, %f format specifier, 44
prefix form, increment operator, 137
for loop, 140
while loop, 152
preprocessing, 529–535
conditional compilation, 536–537
debugging, 539–543
directives testing for specific values, 537
duplicating blocks of code, 536
external variables and functions, 530–531
including header files in programs, 530
joining two results of a macro expansion, 535
macro substitutions, 532
macros that look like functions, 532–534
multiple macro substitutions, 536
standard preprocessing macros, 538
strings as macro arguments, 534–535
substitutions in program source code, 531–532
preprocessor directives, 8
#define, 48
#elif, 538
#else, 537–538
#endif, 536
#if, 536–537
#ifdef, 537
#ifndef, 537
#include, 8
#undef, 537
example using, 17
identifying, 529
logical preprocessor directives, 536–538
multiple lines of, 534
writing Reversi game program, 356
prime numbers
calculating, 264–268, 487–495
adjustment for calloc function, 268
printers
fprintf function, 405–406
writing to printers, 405–406
printf function, 10
see also fprintf function; sprintf function; wprintf function
arguments, 25, 394
arithmetic expressions as arguments, 34
splitting long arguments over 2+ lines, 34
calculating circumference/area of circle, 47
calculating tree height program, 75
character output, 60
coping with mixture of types, 348
escape sequences, 396–397
example using, 16, 17
example using functions, 306, 307
format conversion specifiers, 395
format strings, 394
formatted output to screen, 394–403
character output, 401–403
integer output, 397–399
outputting floating-point values, 400–401

general form, 394
long double types, 52
output format specifiers, 573
output streams, 373
parameters, 394
specifying floating-point decimal places, 44
specifying floating-point field width, 44
string constants, 203
printing characters
function testing for, 110, 222
wide character strings, 229
processor clock, 549
product function, 543
program elements, inspecting, 539
program examples
calculating average of values, 142–144, 175–177
averaging numbers with arrays, 178–180
calculating circumference/area of circle, 45–47
calculating discounted prices, 100–101
calculating factorial of integer, 343–345
calculating hat size, 190–193
using multidimensional arrays and pointers, 261–262
calculating prime numbers, 264–268, 487–495
adjustment for calloc function, 268
calculating tree height, 72–79
calculator, 122–126
using pointers, 283–293
converting uppercase to lowercase, 93–96, 98–99
counting digits and letters entered from keyboard, 222
determining frequency of words in text, 231, 233–238
dividing cookies equally among children, 35–36
drawing box, 134–135
drawing vertical bar chart, 454–464
file-viewer, 522–527
limiting user input, 144–146
memory-test game, 160–172
noughts and crosses game, 194–201
reading text of arbitrary length from keyboard, 231–233
reading/writing/updating binary file, 517–521
recording when program last compiled, 538
Reversi, 353–371
reversing digits of integer, 158–159
selection by criteria, 104–106
Simple Simon program, 160–172
summing numbers
for loop, 138–140
nested for loop, 153–154
while loop, 151–152
while loop nested inside for loop, 154–156
tic-tac-toe game, 194–201
using pointers, 255–258
program flow, tracing, 539
program libraries
see libraries
program structure, 295–299
execution of multi function program, 296
variable scope, 296–299
programming basics
see also C programs
approaching the problem, 173
Build option, 3
compiling source code, 2
decision making, 81
multiple-choice questions, 106
editing source code, 1–2
word processors, 2
exe extension, 3
executing programs, 3
indentation of program code, 173
integrated development environment, 1
linking, 3
looping, 129
modular programming, 14–16
pointers, 218
running program from command line, 5
source code, 1
structured programming, 354
pseudo-random numbers, 147
punctuation characters
determining frequency of words in text, 231, 233
function testing for, 110, 222
wide character strings, 229
put_primes function, 487, 491, 494
putc function, 474, 476
putchar function, 404
puts function, 403–404

Find it faster at http://superindex.apress.com

Q

question mark
 escape sequence for, 13
quotation marks
 escape sequences for, 13

R

r (\r) escape sequence, 12
 common escape sequences, 397
 memory-test game program, 167, 169
 writing to printers, 406
r mode, 469, 471, 476, 521
r+ (r+t)(rt+) text mode, 484, 521
RAM (random access memory), 21, 22
rand function, 146–148
 memory-test game program, 164, 166
 seed number, 147
 setting limits, 147
 using in random macro, 542
RAND_MAX, 147, 542
random numbers, 147
 generating pseudo-random integers, 146–148
random() macro, 542, 543
range limits
 floating-point types, 51
 integer types, 50, 51
 variable types and value ranges, 79
rb mode, 486, 521
rb+ (r+b) mode, 486, 503, 521
read operations
 see also input
 format specifiers, 47
 from text files, 474–476
 opening text files for, 469
 prefixing variable name with &, 47
 strings from text files, 477–480
read-only memory (ROM), 21
readrecord function, 508, 511–512, 514
realloc function, 270–271
records
 listing file contents, 513–514
 reading record from file, 511–512
 reading record from keyboard, 509–510
 updating existing file contents, 514–517
 writing record to file, 510–511
recursion, 343–345
related function
 pointers to structures, 437
 prototype, 437
relational operators
 see comparison operators
remainder operator
 see modulus operator
remove function, 472, 476
rename function, 471
repeating statements
 see loops
reserved words
 see keywords
restrict keyword, 353
return statement, 9, 304–305
return types, 303
return value type, 9
 specifying, 303–304
return values, 300
 caution returning address of local variable, 324
 description, 327
 expressions, 301
 functions not returning values, 300
 global variables replacing, 342
 passing arguments and return value, 309
 structures as function return value, 432–435
Reversi program, 353–371
 basic logic, 355
 best_move function, 368
 computer_move function, 365, 369
 display function, 357, 358
 get_score function, 367
 make_move function, 365
 starting position and initial moves in, 354
 tasks in the program, 355
 valid_moves function, 360
rewind function, 482, 484
right node, binary trees, 440
right-justified values, output flags, 395
ROM (read-only memory), 21
root node, binary trees, 439, 441
rt text mode, 484

S

S conversion type
 output format specifiers, 575
 wide character strings, 227
s conversion type
 input format specifiers, 378, 577
 output format specifiers, 396, 575
 outputting null-terminating string, 206
 outputting singular/plural form, 101
 reading characters using scanf(), 386

scanf function, 376–388
see also sscanf function
accessing structure members, 413
arrays of structures, 416
address of (&) operator, 46, 47, 182
arguments, 376, 388
calculating circumference/area of circle, 46
calculating tree height program, 75
cause of error stopping input, 376
character input, 60
characters in input format string, 382–383
common mistakes using, 388
comparing strings, 217
description, 46
example using functions, 306, 307
format control string parameter, 376
format control strings, 376–381
whitespace characters in, 376
format specifier, 377
formatted input, 375, 376
ignoring characters, 377
input format specifiers, 576
input streams, 373
invalid characters in input stream, 381
keyboard buffer, 170
problem reading strings with, 223
prototype, 376
reading characters, 386–387
reading hexadecimal and octal values, 384–386
reading multiple data items, 123
reading strings, 386, 388
return value, 376
using pointers with scanf(), 249
variable/pointer relationship, 249
variations on floating-point input, 383–384
writing calculator program using pointers, 284
wscanf function, 376
scope
automatic variables, 338
global scope, 489
global variables, 340
static variables, 339, 340
structures declared within structures, 421–422
variable scope, 296–299
screens
formatted output to, 394–403
character output, 401–403
escape sequences, 396–397
integer output, 397–399
outputting floating-point values, 400–401
using printf(), 394–396
unformatted output to, 404
searching strings, 218–221
for substrings, 220–221
strchr function, 219–220
strrchr function, 220
seed number
memory-test game program, 166
rand function, 147
SEEK_CUR/SEEK_END/SEEK_SET, 496
semicolon (;)
basic C syntax, 5
common syntax errors, 6
variable declarations, 25, 27
sentinel value, 349
set_ancestry function, 436, 437
shift bitwise operators, 116, 118
short int type
choosing correct type for variables, 55
common function return types, 303
format specifier for, 47
input type conversion modifiers, 377
output type conversion modifiers, 396
range limits for, 50
range of values, 38
using, 40
short keyword, 40
shortcuts (&=, |=, ^=, >>=, <<=)
bitwise operators, 119
shortcuts (+=, -=, *=, /=, %=)
assignment operations, 70–71
SHRT_MIN/SHRT_MAX, 50
siblings function, 431, 432
sign bit, 560
signed char variable type, 38
signed numbers, 560
signed values, output flags, 395
similar triangles, 74
Simple Simon program, 160–172
sin function, 72
single precision floating-point numbers, 42
single quote
escape sequence for, 13
singular words
outputting appropriate singular/plural form, 101

Find it faster at http://superindex.apress.com

sinh function, 72
size_flag format specifier
 input format specifiers, 576
 output format specifiers, 574
size_t integer type, 52
 calculating prime numbers, 266
 finding length of string, 213
sizeof operator, 52–53
 bytes occupied by numeric types, 53
 dynamic memory allocation, 264
 with structures, 419
 finding size of arrays, 186–187
 joining strings, 209
 pointers, 245
 reading strings from text files, 479
 writing binary files, 487
slashes (/ and \)
 basic C syntax, 5
sorting data
 using binary tree, 444–447
source code, 1, 2
 duplicating blocks of code, 536
 substitutions in, 531–532
source file, 2
space character/flag
 output format specifiers, 395, 573
spaces
 isspace function, 390
specifiers
 see also format control strings/format specifiers
 conversion specifier, 26
 format specifier, 43
 input format specifiers, 377, 576–578
 output format specifiers, 395–396, 573–575
sprintf function
 formatted output to arrays, 404–405
 output format specifiers, 573
 updating binary files, 506
sqrt function, 71, 548
srand function, 147
 memory-test game program, 164, 166
 using preprocessor directives, 542
sscanf function, 405
 input format specifiers, 576
stacks, 263
standard streams, 374–375
statements
 arithmetic assignment, 25
 arithmetic statements, 31–37
 continue statement, 160
 execution of multi function program, 295
 function declarations, 309
 nested blocks, 301
 return statement, 304–305
static keyword, 339
static variables, 338–340, 342
stdarg.h header file, 345, 351
stdaux stream, 375
stdbool.h header file, 66, 82
 function types included in, 351
stddef.h header file, 213
stderr stream, 375, 483–484
stdin stream, 170
 see also getchar function; gets function; scanf function; wscanf function
 associated device, 375
 changing source, 375
 fflush function, 389
 input from keyboards, 375–394
stdio.h header file, 172, 352
 description, 8
 example using, 17
 function types included in, 351
 scanf function, 46
 standard streams, 374
stdlib.h header file, 146, 147, 352
 converting strings to numerical values, 227
 function types included in, 351
 writing calculator program using pointers, 287
stdout stream
 associated device, 375
 description, 170
 formatted output to screens, 394–403
 reassigning, 375
 stderr stream compared, 375
stdprn stream, 375
str_in function, 319–320
 passing data to function using pointers, 318
 returning pointers, 325, 326
str_out function, 322
 incrementing pointers in functions, 326
 passing data to function using pointers, 318
str_sort function, 320–321
 passing data to function using pointers, 318
strcat function, 214–215
 dealing with errors, 484
strchr function, 219–220

strcmp function
comparing strings, 215–218
determining frequency of words in text, 235
sorting strings using pointers, 283
writing calculator program using pointers, 285
strcpy function
copying strings using, 212
determining frequency of words in text, 235
sorting strings using pointers, 282
stream pointers
file pointers, 468, 469, 470
streams
binary streams, 374
character streams, 374
file streams, 468
input streams, 373, 375–394
limit to open streams, 470
nonstandard streams, 374
output streams, 373, 394–403
standard streams, 374–375
stdaux, 375
stderr, 374
stdin, 374
stdout, 374
stdprn, 375
string constants, 203
string data type
how C deals with strings, 203
string functions, 212–221
string.h header file, 352
function types included in, 351
searching strings, 218–221
string library functions, 212
strings
appending, 208–210
wide character strings, 228
arrays of, 210–212
as macro arguments, 534–535
character conversion functions, 224–226
comparing, 215–218
wide character strings, 228
converting to numerical values, 227, 350
copying
using strcpy(), 212
using strncpy(), 213
wide character strings, 228
defining standard messages, 206
examples of strings in memory, 204
finding length of, 206
strlen function, 208, 213
wide character strings, 228
handling, 205–207
using arrays of pointers, 273–283
using pointers, 271–283
input conversion characters, 378, 577
joining, 208–210
strcat function, 214–215
strncat function, 214
length of, 204
marking end of, 204
output conversion characters, 396, 575
outputting null-terminating string, 206
outputting singular/plural form, 101
reading characters using scanf(), 386
reading
fgets function, 389
from text files, 477–480
gets function, 388–389
scanf function, 386
up to given limit, 276–280
searching, 218–221
for specified character, 219–220
for substrings, 220–221
wide character strings, 228
sorting using pointers, 280–283
storing in arrays, 205
text handling methods, 205–207
using assignment operator to copy, 208
varying output message, 101
wide character strings, 227–230
functions operating on, 228
writing strings to text files, 476–477, 477–480
strlen function, 213
finding length of string, 208
sorting strings using pointers, 282
writing and reading files, 476
writing calculator program using pointers, 285
strncat function, 214
strncmp function, 216
strncpy function, 213
strrchr function, 220
strstr function, 220–221, 225, 226
struct type, 409–420
allocating memory dynamically, 418–420
arrays of structures, 414–416
binary tree nodes, 440

Find it faster at http://superindex.apress.com

common function return types, 304
creating binary tree, 441–443
declaring structures within structures, 421–422
defining data types, 451–453
defining structure types, 411
defining structure variables, 411
defining unnamed structures, 414
pointers to structures as structure members, 422–429
unions as members of structures, 451
using pointers, 417
using structures within structures, 420–421
using typedef, 452–453
structure members
accessing, 411–414
assigning values to, 411
arrays of structures, 414–416
dynamic memory allocation, 418–420
individual members, 421
structures in expressions, 417
description, 410
member selection operator, 411
pointers to structures as, 422–429
doubly linked list, 426–429
linked list, 422–426
structures as members of structures, 420–421
structures declared within structures, 421–422
unions as members of structures, 450–451
structured programming, 354
structures
arrays of, 414–416
as function arguments, 430–431, 436–439
as function return value, 432–435
as union members, 450
binary trees, 439–447
bit-fields in, 429–430
defining structure types, 411
defining structure variables, 411
defining using struct, 409–420
drawing vertical bar chart example, 454–464
dynamic memory allocation for, 418–420
naming conventions, 410
pointers to, 417
as function arguments, 431–432, 436–439
dynamic memory allocation, 418–420
returning pointers to structures, 432–435
structure tag, 409
structures in expressions, 417
tm structure type, 549
typedef and, 452–453
unnamed structures, 414
subarrays, referencing, 257
subnodes, 444
substitutions
macro substitutions, 532
multiple macro substitutions, 536
multiple parameterized substitutions, 532
substitutions in program source code, 531–532
substrings
searching strings for, 220–221
subtraction operator
-= shortcut, 70
arithmetic operators, 32
example demonstrating, 33–34
summing numbers example
summing integers backwards, 140
using for loop, 138–140
using nested for loop, 153–154
using while loop, 151–152
nested inside for loop, 154–156
swap function, 318, 321–322
switch statement, 107–115
break statement, 107, 109
calculator program example, 124
controlling with enumeration, 108
default keyword, 108
writing calculator program using pointers, 290, 293

T

t (\t) escape sequence, 12
common escape sequences, 397
example using, 17
tab
escape sequence for, 12, 17, 397
vertical tab, 13
tag name
struct type, 409
tan/tanh functions, 72
temporary work files, 502–503
creating unique file name, 502–503
ternary operator, 99
test_prime function, 487, 490, 492
testing
debugging, 14
functions without bodies, 300
inspecting program elements, 539

integration testing, 14
modular programming, 14
setting breakpoints, 539
setting watches, 539
tracing program flow, 539
text
arbitrary length input from keyboard, 231–233
text files
see files
text mode, 484–485
binary mode compared, 485, 486
text streams
see character streams
text strings
significance of \ character, 11
string and text handling methods, 205–207
TEXTLEN, 232
tic-tac-toe game, 194–201
using pointers, 255–258
time
see date and time
__TIME__ macro, 538
time function library
date and time function library, 545–555
time function, 147, 546, 547
memory-test game program, 164
time.h header file, 147, 167
time_t type, 548
tm structure type, 549, 552
getting date, 550, 552
getting the day for a date, 553, 554, 555
mmembers structures, 549
TMP_MAX, 503
tmpnam function, 503
tolower function, 95, 224, 365
toupper function, 95, 224, 225
towlower function, 96, 229
towupper function, 96, 229, 230, 403
tracing program flow, 539
trailing zeros, output flags, 395
trees
binary trees, 439–447
triangles
similar triangles, 74
trigonometric functions, 72
two-dimensional arrays
see also multidimensional arrays
accessing array elements, 260–262
getting all values in, 257–258
indirection operator (*), 256, 257
pointers and, 255–258
referencing arrays, 257
tic-tac-toe game using pointers, 255–257
type conversions
see also format control strings/format specifiers
explicit type conversion, 57
implicit type conversion, 57–59
input format specifiers, 377, 378, 577
output format specifiers, 396, 574
typedef keyword
defining data types, 451–453
drawing vertical bar chart example, 454
simplifying code using, 453
structures and, 452–453
types
see also data types
choosing correct type for variables, 54–56
common function return types, 303
defining structure types, 411
specifying return value type, 303–304

U

U conversion type, 40
u conversion type, 52
input format specifiers, 378, 577
output format specifiers, 396, 574
UCHAR_MAX/UINT_MAX, 50
ULLONG_MAX/ULONG_MAX, 50
unary minus operators, 36–37
unary operators, 36, 97
#undef preprocessor directive, 537
enabling/disabling assertions, 544
underscores (_)
naming variables, 28
unformatted input, 375
ungetc function, 390–394
unions
anonymous unions, 450
basic operation of, 448–449
initializing, 450
pointers to, 450
sharing memory between variables, 448–451
structures/arrays as union members, 450
syntax for declaring, 448
unions as members of structures, 450–451
unsigned char variable type, 38
unsigned int constant, 40

Find it faster at http://superindex.apress.com

unsigned int variable type
 common function return types, 304
 input conversion characters, 378
 input format specifiers, 577
 output conversion characters, 396
 output format specifiers, 574
 range of values, 38
unsigned integer types, 38–39
 %u format specifier, 52
 implicit type conversion rules, 58
 range limits for integer types, 50
unsigned keyword, 38
unsigned long int constant, 39, 40
unsigned long int variable type, 39
unsigned long long int variable type, 39
unsigned short int variable type, 38
update modes
 writing to binary files, 503–504
 writing to text files, 471, 484
updatefile function, 514–517
uppercase letters
 converting upper/lower cases, 93–96, 98–99
 function testing for, 110, 222
 wide character strings, 229
users
 limiting user input example, 144–146
USHRT_MAX, 50

V

v (\v) escape sequence, 13
va_arg function, 348, 349
 referencing variable number of arguments, 345, 346
va_end function, 349
 copying va_list type, 348
 referencing variable number of arguments, 345, 347
va_list type, 345, 346
 copying, 348
va_start function, 348
 referencing variable number of arguments, 345, 346
valid_moves function, 360
values
 accessing through pointers, 243–246
 defining constants, 48
 floating-point values, 41
variable argument start
 referencing variable number of arguments, 345
variable declarations
 data types, 25
 declaring variables and pointers, 244–245
 description, 25
 example using integers, 25
 implicit declarations, 30
 single statement declaring multiple variables, 30
variable name, 27
variable scope, 296–299
variable types
 defining structures using struct, 409–420
 pointer to char variable type, 271
 pointers, 242
variables
 accessing values through pointers, 243–246
 arithmetic statements, 31–37
 automatic variables, 296, 338
 choosing correct type for, 54–56
 defining structure variables, 411
 description, 23
 enumerations, 64–66
 explicit type conversion, 57
 external variables and functions, 530–531
 fixing as constant value, 49
 floating-point variables, 42
 global variables, 340–342
 initializing, 30–31
 integer variable types, 38–41
 integers, 24–27
 lifetime, 297
 maintaining within declared scope, 338–340
 memory used storing, 37
 minimum/maximum of, 99
 naming, 27–29
 case sensitivity, 28
 keywords, 25
 number of characters in name, 28
 using underscores (_), 28
 pointers, 241, 249
 sharing memory between, 447–451
 sharing between functions, 340–342
 sizeof operator determining bytes occupied, 53
 static variables, 338–340
 storing Boolean values, 66
 storing integers, 24–27
 storing numbers, 24–37
 using, 29–30

variable types and value ranges, 79
variables in functions, 338–342
vertical bar chart, drawing, 454–464
vertical tab
escape sequence for, 13
void keyword, main(), 9
void types
common function return types, 304
functions not returning values, 300
functions returning values, 301
pointers, 242, 303
return statement, 304
specifying return value type, 303

W

w mode, 469, 521
w+ (w+t)(wt+) text mode, 484, 521
watches, setting, 539
wb mode, 486, 521
wb+ (w+b) mode, 521
updating binary files, 503, 504, 506
writing binary files, 512
wchar.h header file, 228, 229–230
wchar_t data type, 63–64, 227–230, 396
wcscat/wcschr/wcscmp/wcscpy functions, 228
wcslen/wcsncmp/wcsncpy functions, 228
wcsstr function, 228, 230
wctype.h header file, 96, 351, 352
while loop, 149–152
do-while loop, 157–159
ending loop with \n or \0, 272
finding length of string, 207
memory-test game program, 166
postfix increment operator (++), 151
prefix increment operator (++), 152
summing numbers examples, 151–152
nested inside for loop, 154–156
typical usage, 173
whitespace
function testing for, 110, 222
wide character strings, 229
input format specifiers, 576, 578
isspace function, 390
whitespace characters, 396
wide character strings, 227–230
classification functions, 229
fgetws function, 229
functions operating on, 228
L modifier, 227
testing and converting, 229–230
width format specifier
input format specifiers, 576
output format specifiers, 574
width value, %f format specifier, 44
word processors
editing source code with, 2
WORDLEN, 233, 234
words
determining frequency in text, 231, 233–238
list of reserved words, 571
wprintf function
see also printf function
escape sequences, 396–397
format conversion specifiers, 395
formatted output to screen, 394
character output, 401–403
write operations
see also output
opening text files for, 469
writefile function, 508, 512–513
writerecord function, 508, 510–511
writing strings to text files, 476–480
writing to text files, 473–476
wscanf function, 376
format control strings, 376–381
wt text mode, 484

X

X conversion type
input format specifiers, 386, 577
output format specifiers, 396, 575
x conversion type
char as hexadecimal value, 62
input format specifiers, 378, 577
output format specifiers, 396, 575
reading hexadecimal values, 384–386

Z

zero
division by, 123
leading zeros, 398, 574
padding with zeros, 395, 574
trailing zeros, 395

Find it faster at http://superindex.apress.com